(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) EP 0 790 581 A2

(12) EUROPEAN PATENT APPLICATION

(43) Date of publication:
20.08.1997 Bulletin 1997/34

(51) Int Cl.[6]: **G06T 15/00**

(21) Application number: **96308513.9**

(22) Date of filing: **26.11.1996**

(84) Designated Contracting States:
GB SE

(30) Priority: **27.11.1995 US 563033**

(71) Applicant: **SUN MICROSYSTEMS, INC.**
Mountain View, CA 94043 (US)

(72) Inventor: **Hu, Xiao Ping**
Mountain View, California 94043 (US)

(74) Representative: **Hogg, Jeffery Keith et al**
Withers & Rogers
4 Dyer's Buildings
Holborn
London EC1N 2JT (GB)

(54) Method for alpha blending images utilizing a visual instruction set

(57) An image alpha blending method utilizing a parallel processor is provided. The computer-implemented method includes the steps of loading unaligned multiple word components into a processor in one machine instruction, each word component associated with a pixel of an image; alpha blending the multiple word components of different source images and a control image in parallel; and storing the alpha blended multiple word components of a destination image into memory in parallel.

EP 0 790 581 A2

Printed by Jouve, 75001 PARIS (FR)

Description

COPYRIGHT NOTICE

RELATED APPLICATIONS

The present invention is related to U.S. Patent Application No. 08/236,572 by Van Hook et al., filed April 29, 1994, entitled "A Central Processing Unit with Integrated Graphics Functions," as well as U.S. Patent Application No. 08/_____,_____ (Atty Dckt No. P-1867) by Chang-Guo Zhou et al., filed March 3, 1995, entitled "Color Format Conversion in a Parallel Processor," both of which are incorporated in their entirety herein by reference for all purposes.

APPENDIX

The appendix is a copy of the "Visual Instruction Set User's Guide."

BACKGROUND OF THE INVENTION

1. Field of the Invention.

The present invention relates generally to image processing and more particularly to blending two images to form a destination image.

2. Description of the Relevant Art.

One of the first uses of computers was the repetitious calculations of mathematical equations. Even the earliest of computers surpassed their creators in their ability to accurately and quickly process data. It is this processing power that make computers very well suited for tasks such as digital image processing.

A digital image is an image where the pixels are expressed in digital values. These images may be generated from any number of sources including scanners, medical equipment, graphics programs, and the like. Additionally, a digital image may be generated from an analog image. Typically, a digital image is composed of rows and columns of pixels. In the simplest gray-scale images, each pixel is represented by a luminance (or intensity) value. For example, each pixel may be represented by a single unsigned byte with a range of 0-255, where 0 specifies the darkest pixel, 255 specifies the brightest pixel and the other values specify an intermediate luminance.

However, images may also be more complex with each pixel being an almost infinite number of chrominances (or colors) and luminances. For example, each pixel may be represented by four bands corresponding to R, G, B, and α. As is readily apparent, the increase in the number of bands has a proportional impact on the number of operations necessary to manipulate each pixel, and therefore the image.

Blending two images to form a resulting image is a function provided by many image processing libraries, for example the XIL imaging library developed by SunSoft division of Sun MicroSystems, Inc. and included in Solaris operating system.

An example of image blending will now be described with reference to Figs. 1A-D. In the simplest example, the two source images (src1 and src2) are blended to form a destination image (d). The blending is controlled by a control image (a) the function of which is described below. All images are 1000 x 600 pixels and src1, src2, and d are one banded grey-scale images.

Referring to Figs. 8A-D, the src2, src1, destination, and control images are respectively depicted where src1 is a car on a road, src2 is a mountain scene, and d is the car superimposed on the mountain scene. Each pixel in the d image is computed from corresponding pixels in the src1, src2, dst images according to the following formula:

$$dst = a*src1 + (1-a)*src2 \qquad \text{Eq.1}$$

where a is either 0, 1, or a fraction. The a values are derived from pixels in the control image which correspond to pixels in src1 and src2. Thus, the calculations of Eq. 2 must be performed for each pixel in the destination image.

Thus, referring to Figs. 8A-D, the values of all pixels in the control image corresponding to the pixels in src1 representing the car is "1" and the value of all pixels in the control image outside the car is "0". Thus, according to Eq. 1, the pixels' values in the destination image corresponding to the "1" pixels in the control image would be represent the car and the pixels in the destination image corresponding to the "0" pixels in the control image would represent the mountain scene. In practice, the pixel values near the edge of the car would have fractional values to make the edge formed by the car and the mountain scene appear realistic.

While the alpha blending function is provided by existing image libraries, typically the function is executed on a processor having integer and floating point units and utilizes generalized instructions for performing operations utilizing those processors.

However, certain problems associated with alpha blending operations can cause the blending to be slow and inefficient when performed utilizing generalized instructions. In particular, most processors have a memory interface designed to access words aligned along word boundaries. For example, if the word is a byte (8 bits) then bytes are transferred between memory and the processor beginning at address 0 so that all addresses must be divisible by 8. However, image data tends to be misaligned, i.e., does not begin or end on aligned byte addresses, due to many factors including multiple bands. Further, words containing multiple bytes are usually transferred between memory and the processor and standard methods do not take advantage of the inherent parallelism due to the presence of multiple pixels in the registers.

Known image blending techniques basically loop through the image and processing each pixel in sequence. This is a very simple process but for a moderately complex 3000x4000 pixel image, the computer may have to perform 192 million instructions or more. This estimation assumes an image of 3000x4000 pixels, each pixel being represented by four bands and four instructions to process each value or band. This calculation shows that what appears to be a simple process quickly becomes very computationaly expensive and time consuming.

As the resolution and size of images increases, improved systems and methods are needed that increase the speed with which computers may blend images. The present invention fulfills this and other needs.

SUMMARY OF THE INVENTION

The present invention provides innovative systems and methods of blending digital images. The present invention utilizes two levels of concurrency to increase the efficiency of image alpha blending. At a first level, machine instructions that are able to process multiple data values in parallel are utilized. At another level, the machine instructions are performed within the microprocessor concurrently. The present invention provides substantial performance increases in image alpha blending technology.

According to one aspect of the invention, an image alpha blending method of the present operations in a computer system and includes the steps of loading multiple word components of an unaligned image into a microprocessor in parallel, each word component associated with a pixel of an image; alpha blending the multiple word components in parallel; and storing the unaligned multiple word components of a destination image into memory in parallel.

According to another aspect of the invention, a computer program product included a computer usable medium having computer readable code embodied therein for causing loading multiple word components of an unaligned image into a microprocessor in parallel, each word component associated with a pixel of an image; alpha blending the multiple word components in parallel; and storing the unaligned multiple word components of a destination image into memory in parallel

Other features and advantages of the present invention will become apparent upon a perusal of the remaining portions of the specification and drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

Fig. 1 illustrates an example of a computer system used to execute the software of the present invention;

Fig. 2 shows a system block diagram of a typical computer system used to execute the software of the present invention;

Fig. 3 is a block diagram of the major functional units in the UltraSPARC-I microprocessor;

Fig. 4 shows a block diagram of the Floating Point/Graphics Unit;

Fig. 5 is a flow diagram of a partitioned multiply instruction;

Fig. 6 is a flow diagram of a partitioned add instruction;

Figs. 7A-C are a flow diagrams of a partitioned pack instruction;

Figs. 8A-D are depictions of source, destination, and control images;

Fig. 9 is flow chart depicting a preferred embodiment of a method of alpha blending two images, and

Fig. 10 is depiction of the modification of word components effected by the steps in a routine for calculating the destination image.

DESCRIPTION OF THE PREFERRED EMBODIMENT

The following are definitions of some of the terms used herein.

Pixel (picture element) - a small section or spot in an image where the image is on a computer screen, paper, film, memory, or the like.

Byte - a unit of information having 8 bits.

Word - a unit of information that is typically a 16, 32 or 64-bit quantity.

Machine instructions (or code) - binary sequences that are loaded and executed by a microprocessor.

In the description that follows, the present invention will be described with reference to a Sun workstation incorporating an UltraSPARC-I microprocessor and running under the Solaris operating system. The UltraSPARC-I is a highly integrated superscaler 64-bit processor and includes the ability to perform multiple partitioned integer arithmetic operations concurrently. The UltraSPARC-I microprocessor will be described below but is also described in U.S. Application No. 08/236,572 by Van Hook et al, filed April 29, 1994, entitled "A Central Processing Unit with Integrated Graphics Functions," which is hereby incorporated by reference for all purposes. The present invention, however, is not limited to any particular computer architecture or operating system. Therefore, the description the embodiments that follow is for purposes of illustration and not limitation.

Fig. 1 illustrates an example of a computer system used to execute the software of the present invention. Fig. 1 shows a computer system 1 which includes a monitor 3, screen 5, cabinet 7, keyboard 9, and mouse 11. Mouse 11 may have one or more buttons such as mouse buttons 13. Cabinet 7 houses a CD-ROM drive 15 or a hard drive (not shown) which may be utilized to store and retrieve software programs incorporating the present invention, digital images for use with the present invention, and the like. Although a CD-ROM 17 is shown as the removable media, other removable tangible media including floppy disks, tape, and flash memory may be utilized. Cabinet 7 also houses familiar computer components (not shown) such as a processor, memory, and the like.

Fig. 2 shows a system block diagram of computer system 1 used to execute the software of the present invention. As in Fig. 1, computer system 1 includes monitor 3 and keyboard 9. Computer system 1 further includes subsystems such as a central processor 102, system memory 104, I/O controller 106, display adapter 108, removable disk 112, fixed disk 116, network interface 118, and speaker 120. Other computer systems suitable for use with the present invention may include additional or fewer subsystems. For example, another computer system could include more than one processor 102 (i.e., a multi-processor system) or a cache memory.

Arrows such as 122 represent the system bus architecture of computer system 1. However, these arrows are illustrative of any interconnection scheme serving to link the subsystems. For example, a local bus could be utilized to connect the central processor to the system memory and display adapter. Computer system 1 shown in Fig. 2 is but an example of a computer system suitable for use with the present invention. Other configurations of subsystems suitable for use with the present invention will be readily apparent to one of ordinary skill in the art.

Fig. 3 is a block diagram of the major functional units in the UltraSPARC-I microprocessor. A microprocessor 140 includes a front end Prefetch and Dispatch Unit (PDU) 142. The PDU prefetches instructions based upon a dynamic branch prediction mechanism and a next field address which allows single cycle branch following. Typically, branch prediction is better than 90% accurate which allows the PDU to supply four instructions per cycle to a core execution block 144.

The core execution block includes a Branch Unit 145, an Integer Execution Unit (IEU) 146, a Load/Store Unit (LSU) 148, and a Floating Point/Graphics Unit (FGU) 150. The units that make up the core execution block may operate in parallel (up to four instructions per cycle) which substantially enhances the throughput of the microprocessor. The IEU performs the integer arithmetic or logical operations. The LSU executes the instructions that transfer data between the memory hierarchy and register files in the IEU and FGU. The FGU performs floating point and graphics operations.

Fig. 4 shows a block diagram of the Floating Point/Graphics Unit. FGU 150 includes a Register File 152 and five functional units which may operate in parallel. The Register File incorporates 32 64-bit registers. Three of the functional units are a floating point divider 154, a floating point multiplier 156, and a floating point adder 158. The floating point units perform all the floating point operations. The remaining two functional units are a graphics multiplier (GRM) 160 and a graphics adder (GRA) 162. The graphical units perform all the graphics operations of the Visual Instruction Set (VIS) instructions.

The VIS instructions are machine code extensions that allow for enhanced graphics capabilities. The VIS instructions typically operate on partitioned data formats. In a partitioned data format, 32 and 64-bit words include multiple word components. For example, a 32-bit word may be composed of four unsigned bytes and each byte may represent a pixel intensity value of an image. As another example, a 64-bit word may be composed of four signed 16-bit words and each 16-bit word may represent the result of a partitioned multiplication.

The VIS instructions allow the microprocessor to operate on multiple pixels or bands in parallel. The GRA performs single cycle partitioned add and subtract, data alignment, merge, expand and logical operations. The GRM performs three cycle partitioned multiplication, compare, pack and pixel distance operations. The following is a description of

some these operations that may be utilized with the present invention.

Fig. 5 is a flow diagram of a partitioned multiply operation. Each unsigned 8-bit component (i.e., a pixel) 202A-D held in a first register 202 is multiplied by a corresponding (signed) 16-bit fixed point integer component 204A-D held in a second register 204 to generate a 24-bit product. The upper 16 bits of the resulting product are stored as corresponding 16-bit result components 205A-D in a result register 206.

Fig. 6 is a flow diagram of a partitioned add/subtract operation. Each 16-bit signed component 202A-D held in the first register 202 is added/subtracted to a corresponding 16-bit signed component 204A-D held in the second register to form a resulting 16-bit sum/difference component which is stored as a corresponding result component 205A-D in a result register 205.

Fig. 7 is a flow diagram of a partitioned pack operation. Each 16-bit fixed value component 202A-D held in a first register 202 is scaled, truncated and clipped into an 8-bit unsigned integer component which is stored as a corresponding result component 205A-D in a result register 205. This operation is depicted in greater detail in Figs. 7B-C.

Referring to Fig. 7B, a 16-bit fixed value component 202A is left shifted by the bits specified by a GSR scale factor held in GSR register 400 (in this example the GSR scale factor 10) while maintaining clipping information. Next, the shifted component is truncated and clipped to an 8-bit unsigned integer starting at the bit immediately to the left of the implicit binary point (i.e., between bits 7 and 6 for each 16-bit word). Truncation is performed to convert the scaled value into a signed integer (i.e., round to negative infinity). Fig. 7C depicts an example with the GSR scale factor equal to 8.

ALPHA BLENDING

Fig. 9 is a flow chart depicting a preferred embodiment of a method of alpha blending two images. In Fig. 9, multi-component words, with each component associated with a pixel value, are loaded from unaligned areas of memory holding the src1, src2, and control images. The components of these multi-component words are processed in parallel to generate components of a multi-component word holding pixel values of the destination image. The components of a destination word are stored in parallel to an unaligned area of memory.

Accordingly, except for doing different arithmetic as dictated by specified precision values and definitions of partitioned operations, each of the routines described below, for each line of pixels, loops through the data by doing, load data, align data, perform arithmetic in parallel, before and after the loop, deal with edges.

Loading Misaligned Image Data

The use of the visual instruction set to load the src1, src2, and alpha images into the registers of the GPU will now be described. For purpose of illustration it is assumed that the src1 image data begins at Address 16005, src2 begins at Address 24003, and dst begins at 8001. Accordingly, neither src1, src2, or dst begins on an aligned byte address. In this example, all images are assumed to comprise 8-bit pixels and have only a single band.

For purposes of explanation the VIS assembly instructions are used in function call notation. This implies that memory locations instead of registers are referenced, hence aligned loads will implied rather than explicitly stated. This notation is routinely used by those skilled in the art and is not ambiguous.

The special visual instructions utilized to load the misaligned data are **alignaddr(addr, offset)** and **falingndata (data_hi,data_lo)**. The function and syntax of these instructions is fully described in Appendix B. The use of these instructions in an exemplary subroutine for loading misaligned data will be described below.

The function of the **alignaddr()** instruction is to return an aligned address equal to the nearest aligned address occurring before the address of misaligned address and write the offset of the misaligned address from the returned address to the GSR. The integer in the offset argument is added to the offset of the misaligned address prior to writing the offset to the GSR.

For example, as stated above, if the starting Address for src1 is 16005, the **alignaddr(16005,0)** returns the aligned address of 16000 and writes 5 to the GSR.

The function of **falignadata()** is to load an 8-byte doubleword beginning at a misaligned address. This is accomplished by loading two consecutive double words having aligned addresses equal to data_hi and data_lo and using the offset written to the GSR to fetch 8 consecutive bytes with the first byte offset from the aligned boundary by the number written to the GSR.

If **s1** is the address of the first byte of the first 64 bit word of the misaligned src1 data then the routine:

```
s1_aligned = alignaddr(s1, 0)
u_s1_0 = s1_aligned[1]
u-s1-1 = s1_aligned[2]
dbl_s1 = falignadata(u_s1_0, u_s1_1)
```

sets **s1_aligned** to the aligned address preceding the beginning to the first 64 bit word, i.e., 16000, sets **u_s1_0** equal to the first word aligned address and **u_s1_1** equal to the second word aligned address, i.e., 160000 and 16008, and returns the first misaligned word of src1 as **dbl_s1**.

This routine can be modified to return the misaligned pixels from src1, **dbl_s2**, and from the control image, **quad_a**, and included in a loop to load all the pixels of src1, src2, and the control image.

Calculating the Destination Image

1. Pixel Length of Src1, Src2, and Control Image is 1 Byte (8 bits)

In this example the pixels, α, in the control image are 8-bit unsigned integers having values between 0-255. Thus, Eq. 1 is transformed into:

$$dst = \alpha/256*src1 + (1-\alpha/256)*src2 \qquad \text{Eq. 2.}$$

or the destination image can be calculated from:

$$dst = src2 + (src1 - src2)*\alpha/256 \qquad \text{Eq. 3}$$

The following routine utilizes visual instructions that provide for parallel processing of 4 pixel values per operation with each pixel comprising 8 bits. Additionally, as will become apparent, the routine eliminates the requirement of explicitly dividing by 256 thereby reducing the processing time and resources required to calculate the pixel values in the destination image.

ROUTINE 2

```
dbl_s1_e = fexpand(read_hi(dbl_s1));
dbl_s2_e = fexpand(read_hi(dbl_s2));
dbl_tmp2 = fsub16(dble_s2_e, dbl_s1_e);
dbl_tmp1 = fmul8x16(read_hi(quad_a), dbl_tmp_2);
dbl_sum1 = fpadd16(dbl_s1_e, dbl_tmp1);

dbl_s1_e = fexpand(read_lo(dbl_s1));
dbl_s2_e = fexpand(read_lo(dbl_s2));
dbl_tmp2 = fsub16(dble_s2_e, dbl_s1_e);
dbl_tmp1 = fmul8x16(read_lo(quad_a), dbl_tmp_2);
dbl_sum2 = fpadd16(dbl_s1_e, dbl_tmp1);
dbl_d =freg_pair(fpack16(dbl_sum1, dbl_sum2)
```

The functions of the various instructions in routine 2 to calculate the pixel values in the destination image will now be described. The variables **dbl_s1**, **dbl_s2**, and **quad_a** are all 8-byte words including 8 pixel values. As will be described more fully above, each byte may be a complete pixel value or a band in a multiple band pixel.

Fig. 10 depicts the modifications to the word components for each operation in the routine. As depicted in Fig. 10, the function of **fexpand(read_hi(dbl_s1)** is to expand the upper 4-bytes of **dbl_s1** into a 64 bit word having 4 16-bit partitions to form **dbl_s1_e**. Each 16-bit partition includes 4 leading 0's, the a corresponding byte from **dbl_s1** and 4 trailing 0's. The variable **dbl_s2_e** used to calculate **dbl- sum1** is similarly formed and the variables **dbl_sl_e** and **dbl_s2_e** used to calculate **dbl_sum2** are formed by expanding the lower 4 bytes of the corresponding variables.

The function of **fsub16(dbl_s2 , dbl_s1_e)** is to calculate the value (src2 - src1). This instruction performs partitioned subtraction on the 4 16-bit components of **dbl_s2** and **dbl_s1** to form **dbl_tmp2**.

The function of **fmul8x16(read_hi(quad_a, dbl_tmp2))** is to calculate the value $\alpha/256*$(src2 - src1). This instruction performs partitioned multiplication of the upper 4 bytes of **quad_a** and the 4 16-bit components of **dbl_tmp2** to form a 24-bit result and truncates the lower 8 bits to form a 16 bit result. Note that the upper 4 bits of each 16 bit component of **dbl-tmp2** are 0's, because of the **fexpand** operation, the lower 4 bits of the 24 bit product are also 0's. The middle 16 bits are the result of multiplying the byte expanded from **dbl-s1** and the corresponding byte from **quad_a** to form the product of $\alpha*$(src2 - src1). Thus, the truncation of the lower 8 bits removes the lower 4 0's, resulting from the previous expansion, and the lower 4 bits of the product to effect division by 256 to form **dbl_tmp1** equal to $\alpha/256*$

(src2 - src1).

The function of **fpadd16(dbl_s1_e, dbl_tmp1)** is to calculate α/256*(src2 - src1) + src2. This instruction performs partitioned addition on the 16-bit components of its arguments.

The function of **freg_pair(pack16(dbl_sum1, pack 16(dbl_sum2)** is to return an 8-byte (64 bits) including 8 pixel values of the destination image. The instruction **fpack16** packs each 16-bit component into a corresponding 8-bit component by left-shifting the 16-bit component by an amount specified in the GSR register and truncating to an 8-bit value. Thus, the left shift removes the 4 leading 0's and the lower bits are truncated to return the 8 significant bits of the destination in each of the 4 components. The function of **freg-pair** is to join the two packed 32 bit variables into a 64-bit variable.

2. Pixel Length of Src1, Src2, and Control Image is 16 bits.

For 16 bits a second routine is utilized which takes into account different requirements of precision.

ROUTINE 2A

```
dbl_halfshort = 0x80008000
dbl_mask_255 = 0x00ff00ff

compute (1 -r)*s1

dbl_a = fsub16(dbl_a, dbl_halfshort);
(void) alignaddr(d_aligned, seven);
dbl_tmp1 = faligndat(dbl_mask_255, dbl_a);
dbl_tmp2 = fand(dbl_tmp1, dbl_mask_255);
flt_hi = fpack16(dbl_tmp2);

dbl_tmp2 = fmul8ulx16(dbl_a, dbl_s1);
dbl_tmp1 = fmul8x16(flt_hi, dbl_s1);
dbl_sum2 = fpadd16(dbl_tmp1, dbl_tmp2);
dbl_sum1 = fpsub16(sbl_s1, dbl_sum2);

compute r*s2

dbl_tmp2 = fmul8ulx16(dbl_a, dbl_s2);
dbl_tmp1 = fmul8x16(flt_hi_a, dbl_s2);
dbl_sum2 = fpadd(dbl_tmp1, dbl_tmp2);

  dbl_d = fpadd16(dbl_sum1, dbl_sum2);
```

Except for **fmu18ulx16** and **fand** the operations in routine 2A are the same as in routine 2 modified to operate on 16 bit components. The description of the functions of those operations will not be repeated.

The function of **fmul18ulx16(dbl_a, dbl_s2)** is to perform it to perform a partitioned multiplication the unsigned lower 8 bits of each 16-bit component in the arguments and return the upper 16 bits of the result for each component as 16-bit components of **dbl-tmp2.**

The function of **fand(dbl_tmp1, dbl_mask_255)** is to perform a logical AND operation on the variables defined by the arguments.

Storing the Misaligned Destination Image Data

1. Loading Utilizing an Edge Mask and Partial Instruction.

The following routine calculates an edge mask and utilizes a partial store operation to store the destination image data where d is a pointer to the destination location, **d_aligned** is the aligned address which immediately precedes **d**, and width is the width of a destination word.

ROUTINE 3

```
d_end = d + width - 1;
emask = edge_8(d, d_end);
```

```
pst_8(dbl_d, (void *)d_aligned, emask);
++d_aligned;
emask = edge8(d_aligned, d_end);
```

The function of **emask** is to generate a mask for storing unaligned data. For example, if the destination data is to be written to address 0x10003, and the previous aligned address in 0x10000, then the **emask** will be [00011111] and the **pst** instruction will start writing at address 0x10000 and **emask** will disable writes to 0x10000, 0x10001, and 0x10002 and enable writes to 0x10003-0x10007. Similarly, after **emask** is incremented the last part of **dbl_d** is written to the 0x10008, 0x10009, and 0x1000A and the addresses 0x1000B-0x1000F will be masked.

2.3.2 *Floating Point/Graphics Unit (FGU)*

The Floating-Point and Graphics Unit (FGU) as illustrated in Figure 2-4 integrates five functional units and a 32 registers by 64 bits Register File. The floating-point adder, multiplier and divider perform all FP operations while the graphics adder and multiplier perform the graphics operations of the Visual Instruction Set.

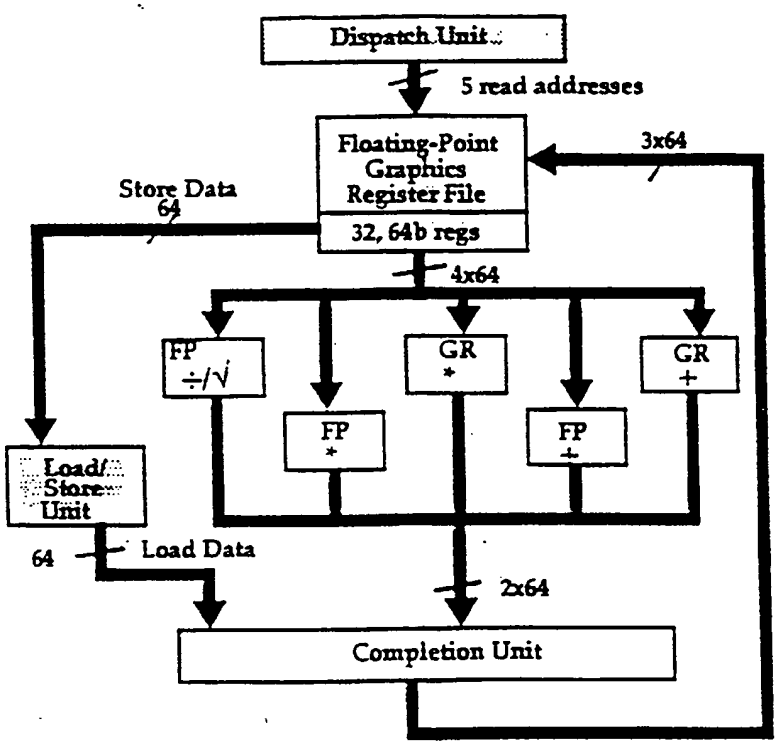


Figure 2-4 Floating Point and Graphics Unit

A maximum of two floating-point/graphics Operations (FGops) and one FP load/store operation are executed in every cycle (plus another integer or branch instruction). All operations, except for divide and square-root, are fully pipelined. Divide and square-root operations complete out-of-order without inhibiting the concurrent execution of other FGops.The two graphics units are both fully pipelined and perform operations on 8 or 16-bit pixel components with 16 or 32-bit intermediate results.

The Graphics Adder performs single cycle partitioned add and subtract, data alignment, merge, expand and logical operations. Four 16-bit adders are utilized and a custom shifter is implemented for byte concatenation and variable byte-length shifting. The Graphics Multiplier performs three cycle partitioned multiplication, compare, pack and pixel distance operations. Four 8x16 multipliers are utilized and a custom shifter is implemented. Eight 8-bit pixel subtractions, absolute values, additions and a final alignment are required for each pixel distance operation.

4.3.3 *vis_freg_pair()*

Function

Join two vis_f32 variables into a single vis_d64 variable.

Syntax

```
vis_d64 vis_freg_pair(vis_f32 data1_32, vis_f32 data2_32);
```

Description

vis_freg_pair() joins two vis_f32 values *data1_32 and data2_32* into a single vis_d64 variable. This offers a more optimum way of performing the equivalent of using vis_write_hi() and vis_write_lo() since the compiler attempts to minimize the number of floating point move operations by strategically using register pairs.

Example

```
vis_f32 data1_32, data2_32;
vis_d64 data_64;

/* Produces data_64, with data1_32 as the upper and data2_32 as the
lower component.*/
data_64 = vis_freg_pair(data1_32, data2_32);
```

4.5 *Pixel Compare Instructions*

4.5.1 *vis_fcmp[gt, le, eq, ne. lt, ge][16,32]()*

Function

Perform logical comparison between two partitioned variables and generate an integer mask describing the result of the comparison.

Syntax

```
int vis_fcmpgt16(vis_d64 data1_4_16, vis_d64 data2_4_16);
int vis_fcmple16(vis_d64 data1_4_16, vis_d64 data2_4_16);
int vis_fcmpeq16(vis_d64 data1_4_16, vis_d64 data2_4_16);
int vis_fcmpne16(vis_d64 data1_4_16, vis_d64 data2_4_16);
int vis_fcmpgt32(vis_d64 data1_2_32, vis_d64 data2_2_32);
int vis_fcmpeq32(vis_d64 data1_2_32, vis_d64 data2_2_32);
int vis_fcmple32(vis_d64 data1_2_32, vis_d64 data2_2_32);
int vis_fcmpne32(vis_d64 data1_2_32, vis_d64 data2_2_32);
int vis_fcmplt16(vis_d64 data1_4_16, vis_d64 data2_4_16);
int vis_fcmplt32(vis_d64 data1_2_32, vis_d64 data2_2_32);
int vis_fcmpge16(vis_d64 data1_4_16, vis_d64 data2_4_16);
int vis_fcmpge32(vis_d64 data1_2_32, vis_d64 data2_2_32);
```

Description

vis_fcmpt[gt, le, eq, neq,lt,ge]() compare four 16 bit partitioned or two 32 bit partitioned fixed-point values within *data1_4_16, data1_2_32* and *data2_4_16, data2_2_32*. The 4 bit or 2 bit comparison results are returned in the corresponding least significant bits of a 32 bit value, that is typically used as a mask. A single bit is returned for each partitioned compare and in both cases bit zero is the least significant bit of the compare result.

For vis_fcmptgt(), each bit within the 4 bit or 2 bit compare result is set if the corresponding value of [*data1_4_16, data1_2_32*] is greater than the corresponding value of [*data2_4_16, data2_2_32*].

For vis_fcmptle(), each bit within the 4 bit or 2 bit compare result is set if the corresponding value of [*data1_4_16, data1_2_32*] is less than or equal to the corresponding value of [*data2_4_16, data2_2_32*.

For vis_fcmpteq(), each bit within the 4 bit or 2-bit compare result is set if the corresponding value of [*data1_4_16, data1_2_32*] is equal to the corresponding value of [*data2_4_16, data2_2_32*].

For vis_fcmptne(), each bit within the 4 bit or 2 bit compare result is set if the corresponding value of [*data1_4_16, data1_2_32*] is not equal to the corresponding value of [data2_4_16, data2_2_32].

For vis_fcmptlt(), each bit within the 4 bit or 2 bit compare result is set if the corresponding value of [data1_4_16, data1_2_32] less than the corresponding value of [data2_4_16, data2_2_32].

For vis_fcmptge() each bit within the 4 bit or 2 bit compare result is set if the corresponding value of [data1_4_16, data1_2_32] is greater or equal to the corresponding value of [data2_4_16, data2_2_32].

The four 16 bit pixel comparison operations are illustrated in Figure 4-4 and the two 32 bit pixel comparison operations are illustrated in Figure 4-5.

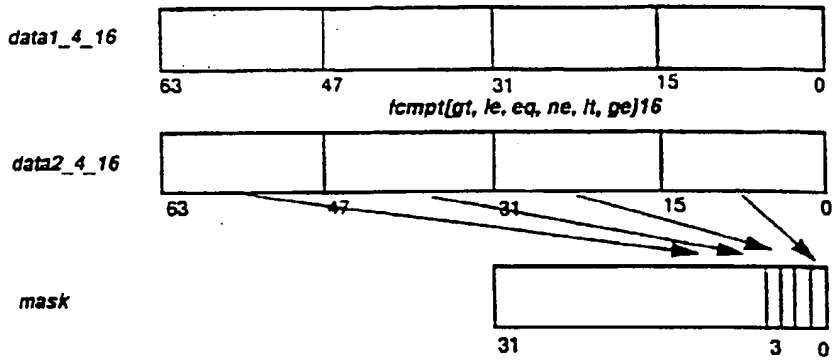


Figure 4-4 Four 16 bit Pixel Comparison Operations

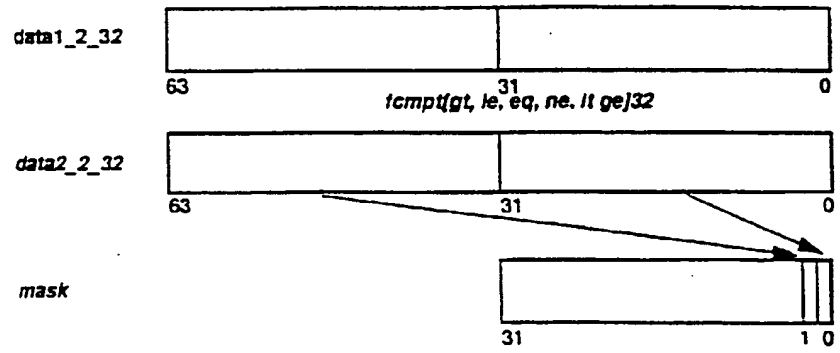


Figure 4-5 Two 32 bit Pixel Comparison Operation

Example

```
int mask;
vis_d64 data1_4_16, data2_4_16, data1_2_32, data2_2_32;

mask = vis_fcmptgt16(data1_4_16, data2_4_16);
/* data1_4_16 > data2_4_16 */

mask = vis_fcmptle16(data1_4_16, data2_4_16);
/* data1_4_16 <= data2_4_16 */

mask = vis_fcmptle16(data1_4_16, data2_4_16);
/* data1_4_16 >= data2_4_16 */

mask = vis_fcmpteq16(data1_4_16, data2_4_16);
/* data1_4_16 = data2_4_16 */

mask = vis_fcmptne16(data1_4_16, data2_4_16);
/* data1_4_16 = data2_4_16 */

mask = vis_fcmplt16(data1_4_16, data2_4_16);
/* data1_4_16 < data2_4_16 */

mask = vis_fcmpgt16(data1_4_16, data2_4_16);
/* data1_4_16 > data2_4_16 */

/* mask may be used as an argument to a partial store instruction
vis_pst_8, vis_pst_16 or vis_pst_32*/
vis_pst_16(data1_4_16, &data2_4_16, mask);
/* Stores the greater of data_1_4_16 or data2_4_16 overwriting
data2_4_16 */
```

4.6 *Arithmetic Instructions*

The VIS arithmetic instructions perform partitioned addition, subtraction or multiplication.

4.6.1 *vis_fpadd[16, 16s, 32, 32s](), vis_fpsub[16, 16s, 32, 32s]()*

Function

Perform addition and subtraction on two 16 bit, four 16 bit or two 32 bit partitioned data.

Syntax:

```
vis_d64 vis_fpadd16(vis_d64 data1_4_16, vis_d64 data2_4_16);
vis_d64 vis_fpsub16(vis_d64 data1_4_16, vis_d64 data2_4_16);
vis_d64 vis_fpadd32(vis_d64 data1_2_32, vis_d64 data2_2_32);
vis_d64 vis_fpsub32(vis_d64 data1_2_32, vis_d64 data2_2_32);
vis_f32 vis_fpadd16s(vis_f32 data1_2_16, vis_f32 data2_2_16);
```

```
vis_f32 vis_fpsub16s(vis_f32 data1_2_16, vis_f32 data2_2_16);
vis_f32 vis_fpadd32s(vis_f32 data1_1_32, vis_f32 data2_1_32);
vis_f32 vis_fpsub32s(vis_f32 data1_1_32, vis_f32 data2_1_32);
```

Description

vis_fpadd16() and **vis_fpsub16()** perform partitioned addition and subtraction between two 64 bit partitioned variables, interpreted as four 16 bit signed components, *data1_4_16* and *data2_4_16* and return a 64bit partitioned variable interpreted as four 16 bit signed components, *sum_4_16* or *difference_4_16*. **vis_fpadd32()** and **vis_fpsub32()** perform partitioned addition and subtraction between two 64 bit partitioned components, interpreted as two 32 bit signed variables, *data1_2_32* and *data2_2_32* and return a 64 bit partitioned variable interpreted as two 32 bit components, *sum_2_32 or difference_2_32*. Overflow and underflow are not detected and result in wraparound. Figure 4-6 illustrates the **vis_fpadd16()** and **vis_fpsub16()** operations. Figure 4-7 illustrates the **vis_fpadd32()** and **vis_fpsub32()** operation. The 32 bit versions interpret their arguments as two 16 bit signed values or one 32 bit signed value.

The single precision version of these instructions **vis_fpadd16s()**, **vis_fpsub16s()**, **vis_fpadd32s()**, **vis_fpsub32s()** perform two 16-bit or one 32-bit partitioned adds or subtracts. Figure 4-8 illustrates the **vis_fpadd16s()** and **vis_fpsub16s()** operation and Figure 4-9 illustrates the **vis_fpadd32s()** and **vis_fpsub32s()** operation.

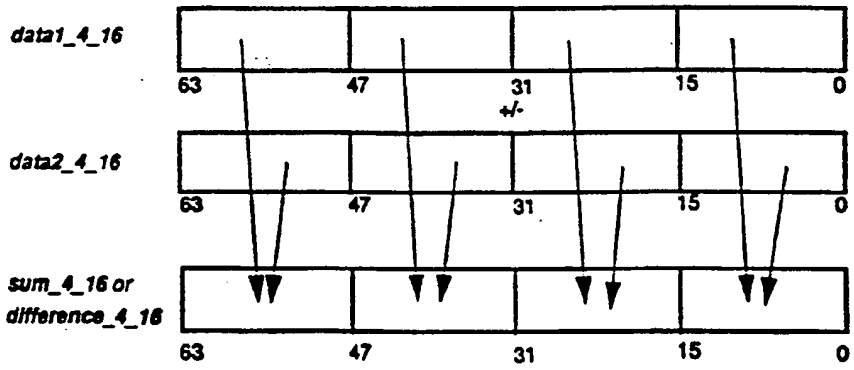


Figure 4-6 vis_fpadd16() and vis_fpsub16() operation

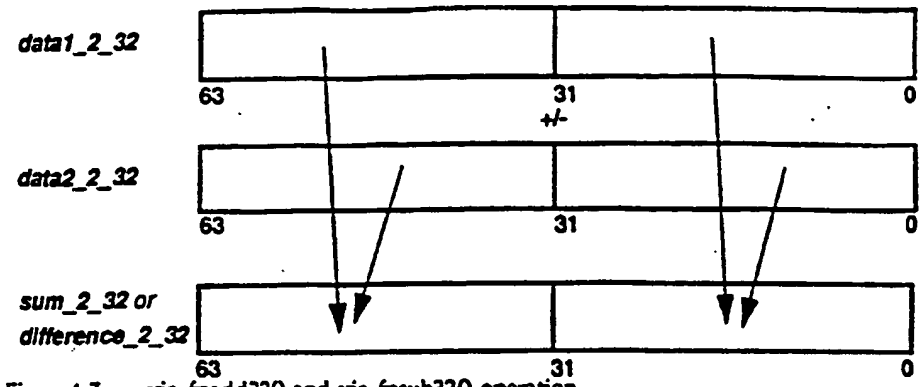


Figure 4-7 vis_fpadd320 and vis_fpsub320 operation

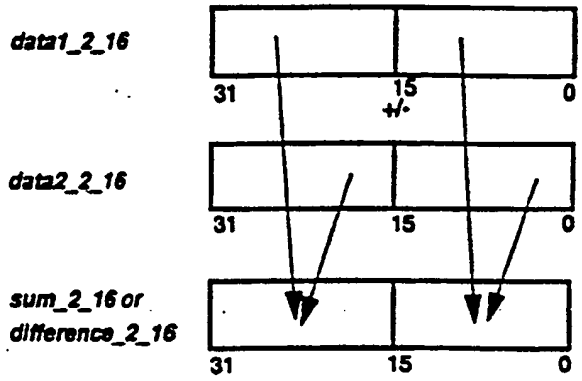


Figure 4-8 vis_fpadd16s() and vis_fpsub16s() operation

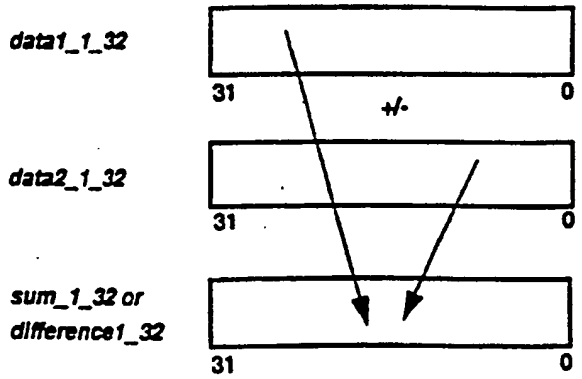


Figure 4-9 vis_fpadd32s() and vis_fpsub32s()

Example

```
vis_d64 data1_4_16, data2_4_16, data1_2_32, data2_2_32;
vis_d64 sum_4_16, difference_4_16, sum_2_32, difference_2_32;
vis_f32 data1_2_16, data2_2_16, sum_2_16, difference_2_16;
vis_f32 data1_1_32, data2_1_32, sum_1_32, difference1_32;

sum_4_16 = vis_fpadd16(data1_4_16, data2_4_16);
difference_4_16 = vis_fpsub16(data1_4_16, data2_4_16);
sum_2_32 = vis_fpsum32(data1_2_32, data2_2_32);
difference_2_32 = vis_fpsub32(data1_2_32, data2_2_32);
sum_2_16 = vis_fpadd16s(data1_2_16, data2_2_16);
difference_2_16 = vis_fpsub16s(data1_2_16, data2_2_16);
sum_1_32 = vis_fpadd32s(data1_1_32, data2_1_32);
difference1_32 = vis_fpsub32s(data1_1_32, data2_1_32);
```

4.6.2 *vis_fmul8x16()*

Function:

Multiply the elements of an 8 bit partitioned vis_f32 variable by the corresponding element of a 16 bit partitioned vis_d64 variable to produce a 16 bit partitioned vis_d64 result.

Syntax:

```
vis_d64 vis_fmul8x16(vis_f32 pixels, vis_d64 scale);
```

Description

vis_fmul8x16() multiplies each unsigned 8-bit component within *pixels* by the corresponding signed 16-bit fixed-point component within *scale* and returns the upper 16 bits of the 24 bit product (after rounding) as a signed 16-bit component in the 64 bit returned value. Or in other words:

16 bit result = (8 bit pixel element*16 bit scale element + 128)/256

The operation is illustrated in Figure 4-10.

This instruction treats the *pixels* values as fixed-point with the binary point to the left of the most significant bit. For example, this operation is used with filter coefficients as the fixed-point *scale* value, and image data as the *pixels* value.

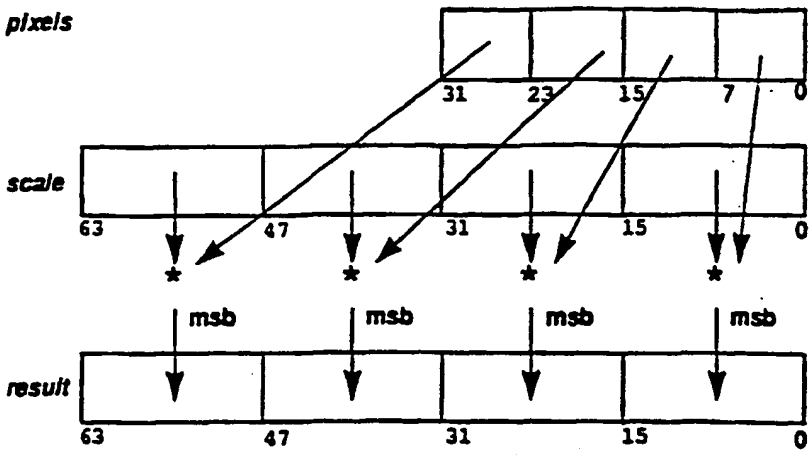


Figure 4-10 vis_fmul8x16() Operation

Example

```
vis_f32 pixels;
vis_d64 result, scale;

result = vis_fmul8x16(pixels, scale);
```

4.6.4 *vis_fmul8sux16(), vis_fmul8ulx16()*

Function

Multiply the corresponding elements of two 16 bit partitioned vis_d64 variables to produce a 16 bit partitioned vis_d64 result.

Syntax

```
vis_d64 vis_fmul8sux16(vis_d64 data1_16, vis_d64 data2-16);
vis_d64 vis_fmul8ulx16(vis_d64 data1_16, vis_d64 data2_16);
```

Description

Both **vis_fmul8sux16()** and **vis_fmul8ulx16()** perform "half" a multiplication. **fmul8sux16()** multiplies the upper 8 bits of each 16-bit signed component of *data1_4_16* by the corresponding 16-bit fixed point signed component in *data2_4_16*. The upper 16 bits of the 24-bit product are returned in a 16-bit partitioned *resultu*. The 24 bit product is rounded to 16 bits . The operation is illustrated in Figure 4-13.

vis_fmul8ulx16() multiplies the unsigned lower 8 bits of each 16-bit element of *data1_4_16* by the corresponding 16 bit element in *data2_4_16*. Each 24-bit product is sign-extended to 32 bits. The upper 16 bits of the sign extended value are returned in a 16-bit partitioned *resultl*. The operation is illustrated in Figure 4-14.

Because the result of **fmul8ulx16()** is conceptually shifted right 8 bits relative to the result of **fmul8sux16()** they have the proper relative significance to be added together to yield a 16 bit product of *data1_4_16* and *data2_4_16*.

Each of the "partitioned multiplications " in this composite operation, multiplies two 16-bit fixed point numbers to yield a 16-bit result. i.e. the lower 16-bits of the full precision 32-bit result are dropped after rounding. The location of the binary point in the fixed point arguments is under user's control. It can be anywhere from to the right of bit 0 or to the left of bit 15.

For example, each of the input arguments can have 8 fractional bits. i.e. the binary point between bit 7 and bit 8. If a full precision 32-bit result were provided, it would have 16 fractional bits. i.e. the binary point would be between bits 15 and 16. Since, however, only 16 bits of the result are provided, the lower 16 fractional bits are dropped after rounding. The binary point of the 16-bit result in this case is to the right of bit 0.

Another example, illustrated below, has 12 fractional bits in each of it's 2 component arguments. i.e. the binary point is between bits 11 and 12. A full precision 32-bit result would have 24 fractional bits. i.e. the binary point between bits 23 and 24. Since, however, only a 16-bit result is provided, the lower 16 fractional bits are dropped after rounding, thus providing a result with 8 fractional bits. i.e. the binary point between bits 7 and 8.

```
    0101.001010010101  (= 5.161376953125)
x   0001.011001001001  (= 1.392822265625)
--------------------
    00000111.00110000  (= 7.188880741596)
```

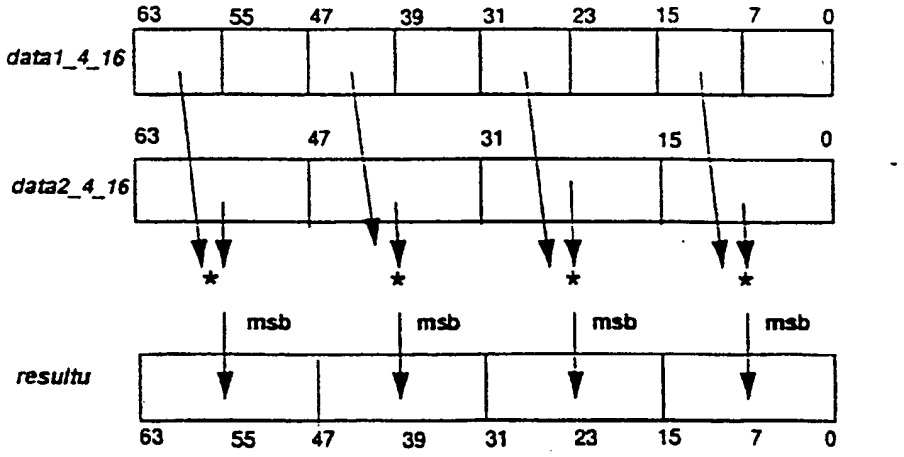


Figure 4-13 vis_fmul8sux16() operation

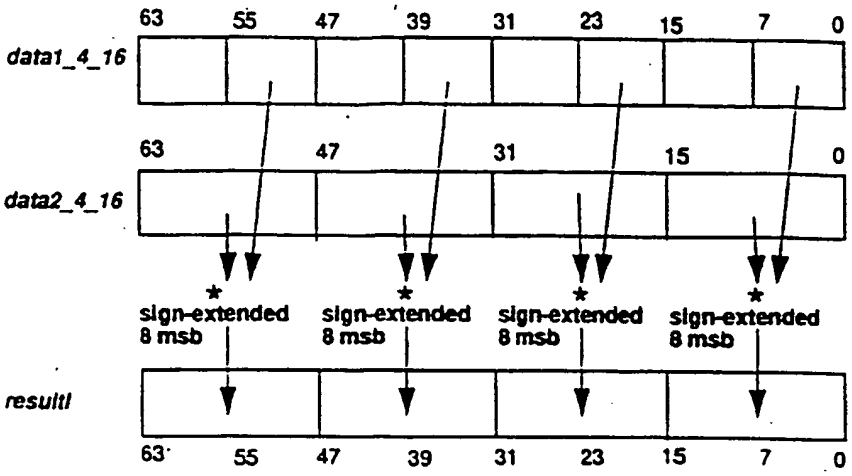


Figure 4-14 vis_fmul8ulx16() operation

Example

```
vis_d64 data1_4_16, data2_4_16, result1, resultu, result;

resultu = vis_fmul8sux16(data8_8, data4_16);
result1 = vis_fmul8ulx16(data8, data16);
result = visfpadd16(resultu, result1); /* 16 bit result of a 16*16
multiply */
```

4.7.1 *vis_fpack16()*

Function

Truncates four 16 bit signed components to four 8 bit unsigned components.

Syntax

```
vis_f32 fpack16(vis_d64 data_4_16);
```

Description

vis_fpack16() takes four 16-bit fixed components within *data_4_16*, scales, truncates and clips them into four 8-bit unsigned components and returns a vis_f32 result. This is accomplished by left shifting the 16 bit component as determined from the scale factor field of GSR and truncating to an 8-bit unsigned integer by rounding and then discarding the least significant digits. If the resulting value is negative (i.e., the MSB is set), zero is

returned. If the value is greater than 255, then 255 is returned. Otherwise the scaled value is returned. For an illustration of this operatioin see 4.7.2.

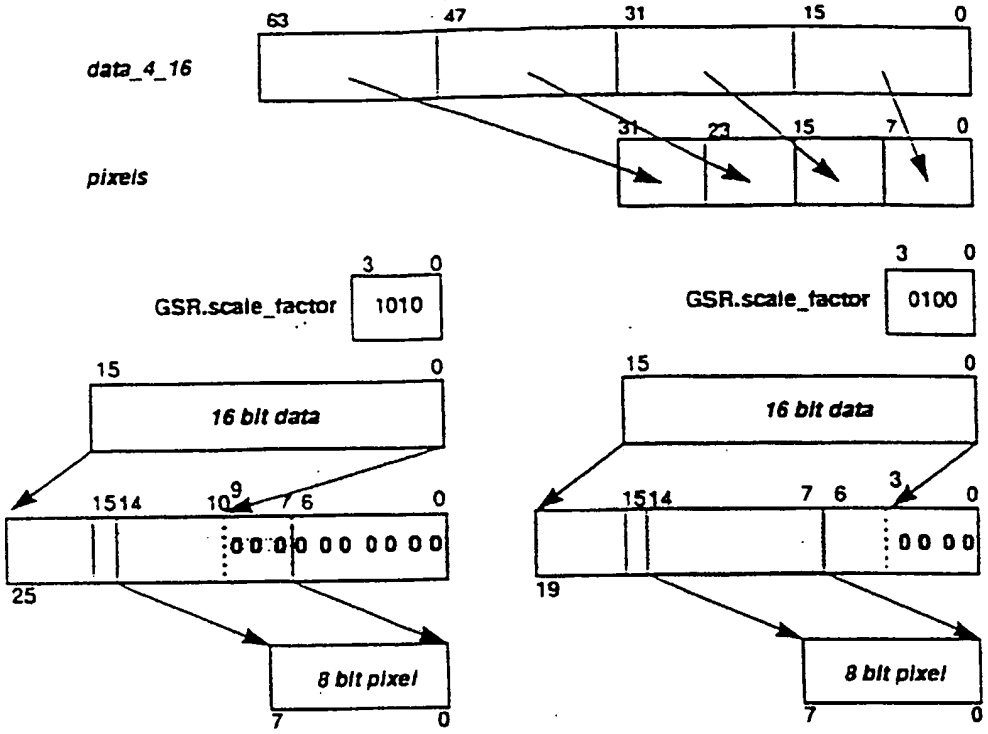


Figure 4-17 vis_fpack16() operation

Example

```
vis_d64 data_4_16;
vis_f32 pixels;

pixels = vis_fpack16(data_4_16);
```

4.7.2 *vis_fpack32()*

Function

Truncate two 32 bit fixed values into two unsigned 8 bit integers.

Syntax

```
vis_d64 vis_fpack32(vis_d64 data_2_32, vis_d64 pixels);
```

Description

vis_fpack32() copies its second argument, *pixels* shifted left by 8 bits into the destination or vis_d64 return value. It then extracts two 8 bit quantities, one each from the two 32-bit fixed values within *data_2_32*, and overwrites the least significant byte position of the destination. Two pixels consisting of four 8 bit bytes each may be assembled by repeated operation of vis_fpack32 on four *data_2_32* pairs.

The reduction of data_2_32 from 32 to 8 bits is controlled by the scale factor of the GSR. The initial 32-bit value is shifted left by the GSR.scale_factor, and the result is considered as a fixed-point number with its binary point between bits 22 and 23. If this number is negative, the output is clamped to 0; if greater than 255, it is clamped to 255. Otherwise, the eight bits to the left of the binary point are taken as the output.

Another way to conceptualize this process is to think of the binary point as lying to the left of bit (22 - scale factor) i.e(., 23 - scale factor) bits of fractional precision. The 4-bit scale factor can take any value between 0 and 15 inclusive. This means that 32-bit partitioned variables which are to be packed using vis_fpack32() may have between 8 and 23 fractional bits.

The following code examples takes four variables red, green, blue, and alpha, each containing data for two pixels in 32-bit partitioned format (r0r1, g0g1, b0b1, a0a1), and produces a vis_d64 pixels containing eight 8 bit quantities (r0g0b0a0r1g1b1a1).

```
vis_d64 red, green, blue, alpha, pixels;
/*red, green, blue, and alpha contain data for 2 pixels*/

pixels = vis_fpack32(red, pixels);
pixels = vis_fpack32(green, pixels);
pixels = vis_fpack32(blue pixels);
pixels = vis_fpack32(alpha, pixels);
/* The result is two sets of red, green, blue and alpha values packed
in pixels */
```

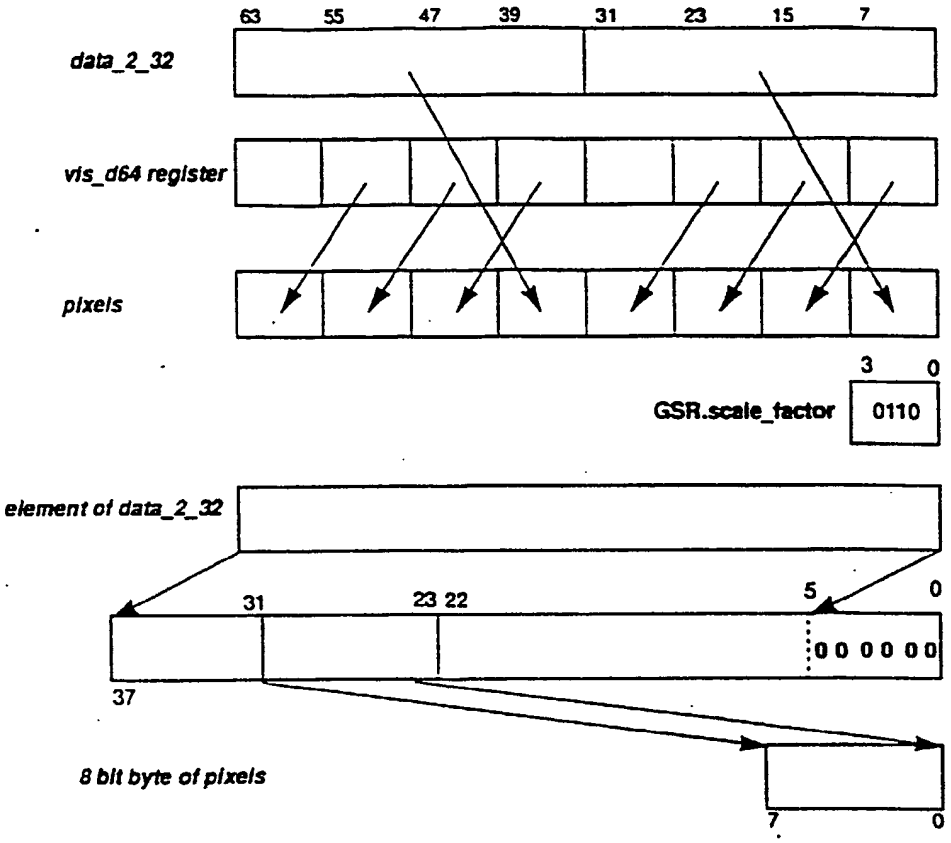


Figure 4-18 vis_fpack32() operation

4.7.3 *vis_fpackfix()*

Function

Converts two 32 bit partitioned data to two 16 bit partitioned data.

Syntax

```
vis_f32 fpackfix(vis_d64 data_2_32,);
```

Description

vis_fpackfix() takes two 32-bit fixed components within *data_2_32*, scales, and runcates them into two 16-bit signed components. This is accomplished by shifting each 32 bit component of *data_2_32* according to GSR.scale-factorand then truncating to a 16 bit scaled value starting between bits 16 and 15 of each 32 bit word. Truncation converts the scaled value to a signed integer (i.e. rounds toward negative infinity). If the value is less than -32768, -32768 is returned. If the value is greater than 32767, 32767 is returned. Otherwise the scaled *data_2_16* value is returned. Figure 4-19 illustrates the vis_fpackfix() operation.

Example

```
vis_d64 data_2_32;
vis_f32 data_2_16;

data_2_16 = vis_fpackfix(data_2_32);
```

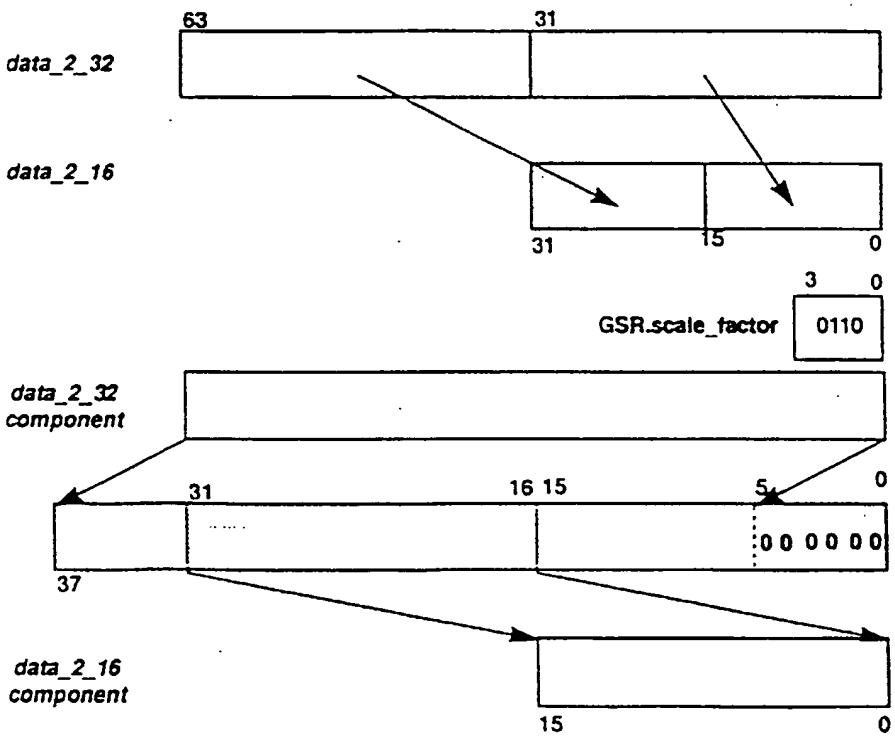


Figure 4-19 vis_fpackfix() operation

4.7.4 *vis_fexpand()*

Description

Converts four unsigned 8 bit elementsto four 16 bit fixed elements.

Syntax

```
vis_d64 fexpand(vis_f32 data_4_8);
```

Description

vis_fexpand() converts packed format data e.g. raw pixel data to a partitioned format. vis_fexpand() takes four 8-bit unsigned elements within *data_4_8*, converts each integer to a 16-bit fixed value by inserting four zeroes to the right and to the left of each byte, and returns four 16-bit elements within a 64 bit result. Since the various vis_fmul8x16() instructions can also perform this function, vis_fexpand() is mainly used when the first operation to be used on the expanded data is an addition or a comparison. Figure 4-20 illustrates the vis_fexpand() operation.

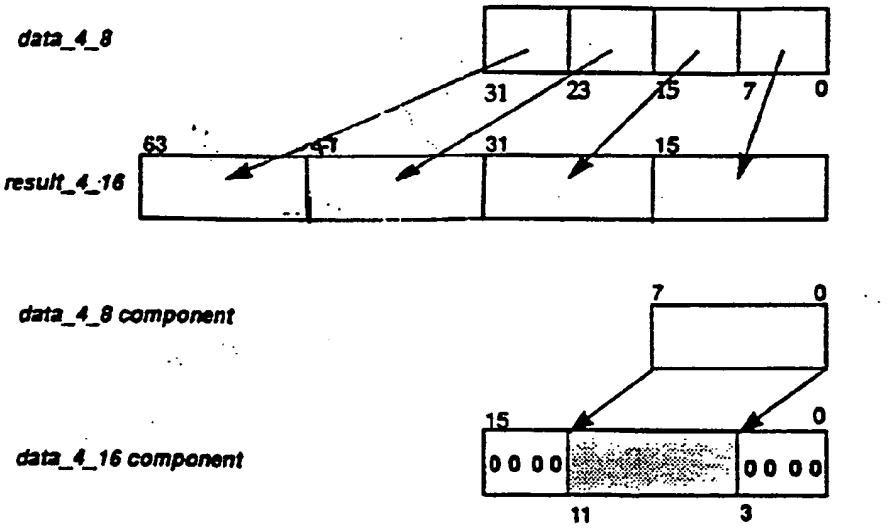


Figure 4-20 vis_fexpand() operation

Example

```
vis_d64 data_4_16, result_4_16;
vis_f32 data_4_8, factor;

result_4_16 = vis_fexpand(data_2_32);
```

```
/*Using vis_fmul8x16al to perform the same function*/
factor = vis_float_(0x0010);
result_4_16 = vis_fmul_8x16al(data_2_32, factor);
```

4.7.7 *vis_alignaddr(), vis_faligndata()*

Function

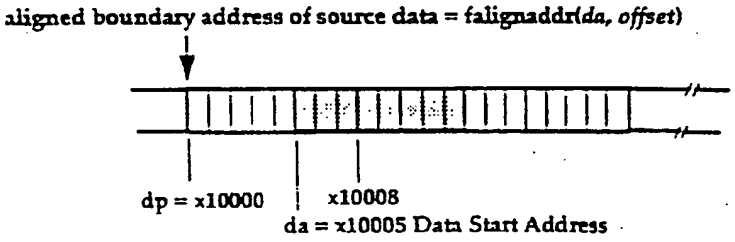
Calculate 8 byte aligned address and extract an arbitrary 8 bytes from two 8 byte aligned addresses.

Syntax

```
void *vis_alignaddr(void *addr, int offset);
vis_d64 vis_faligndata(vis_d64 data_hi, vis_d64 data_lo);
```

Description

vis_alignaddr() and vis_faligndata() are usually used together. vis_alignaddr() takes an arbitrariiy aligned pointer *addr* and a signed integer *offset*, adds them, places the rightmost three bits of the result in the address offset field of the GSR and returns the result with the rightmost 3 bits set to 0. This return value can then be used as an 8 byte aligned address for loading or storing a vis_d64 variable. An example is shown in Figure 4-22.



vis_alignaddr(x10005, 0) returns x10000 with 5 placed in the GSR offset field.

vis_alignaddr(x10005, -2) returns x10000 with 3 placed in the GSR offset field.

Figure 4-22 vis_alignaddr() example.

vis_faligndata() takes two vis_d64 arguments *data_hi* and *data_lo*. It concatenates these two 64 bit values as *data_hi*, which is the upper half of the concatenated value, and *data_lo*, which is the lower half of the concatenated value. Bytes in this value are numbered from most significant to the least significant with the most significant byte being 0. The return value is a vis_d64 variable representing eight bytes extracted from the concatenated value with the most significant byte specified by the GSR offset field as illustrated in Figure 4-23.

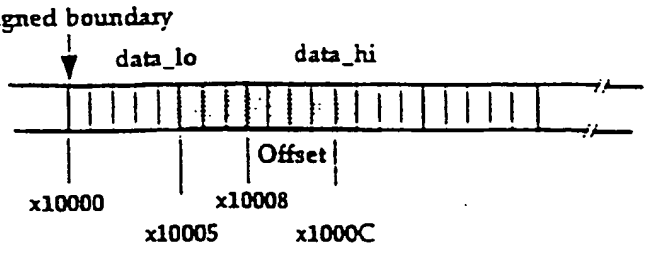


Figure 4-23 vis_faligndata() example.

Care must be taken not to read past the end of a legal segment of memory. A legal segment can only begin and end on page boundaries, and so if any byte of a vis_d64 lies within a valid page, the entire vis_d64 must lie within the page. However, when *addr* is already 8 byte aligned, the GSR alignment bits will be set to 0 and no byte of *data_lo* will be used. Therefore even though it is legal to read 8 bytes starting at *addr*, it may not be legal to read 16 bytes and this code will fail. This problem may be avoided in a number of ways:

- *addr* may be compared with some known address of the last legal byte;
- the final iteration of a loop, which may need to read past the end of the legal data, may be special-cased;
- slightly more memory than needed may be allocated to ensure that there are valid bytes available after the end of the data.

Example

The following example illustrates how these instructions may be used together to read a group of eight bytes from an arbitrarily-aligned address 'addr', as follows:

```
void *addr, *addr_aligned;
vis_d64 data_hi, data_lo, data;

addr_aligned = vis_alignaddr(addr, 0);
data_hi = addr_aligned[0];
data_lo = addr_aligned[1];
data = vis_faligndata(data_hi, data_lo);
```

When data are being accessed in a stream, it is not necessary to perform all the steps shown above for each vis_d64. Instead, the address may be aligned once and only one new vis_d64 read per iteration:

```
addr_aligned = vis_alignaddr(addr, 0);
data_hi = addr_aligned[0];
data_lo = addr_aligned[1];
for (i = 0; i < times; --i) {
    data = vis_faligndata(data_hi, data_lo);
    /* Use data here. */

    /* Move data "window" to the right. */
    data_hi = data_lo;
    data_lo = addr_aligned[i + 2];
}
```

Of course, the same considerations concerning read ahead apply here. In general, it is best not to use vis_alignaddr() to generate an address within an inner loop, e.g.,

```
{
addr_aligned = vis_alignaddr(addr, offset);
data_hi = addr_aligned[0];
offset -= 8;
/* ... */
}
```

Since this means that the data cannot be read until the new address has been computed. Instead, compute the aligned address once and either increment it directly or use array notation. This will ensure that the address arithmetic isperformed in the integer units in parallel with the execution of the VIS instructions.

Claims

1. In a computer system, a method of alpha blending images, comprising the steps of:

loading a first word, comprising a plurality of word components into a processor in parallel, each word component associated with a sourcel pixel of a first source image;
loading a second word comprising plurality of word components into a processor in parallel, each word component associated with a source2 pixel of a second source image;
loading a third word comprising plurality of word components into a processor in parallel, each word component associated with a control pixel of a control image;
alpha blending the components of said first, second, and third words in parallel to generate word components of a fourth, with the word components of said fourth word associated with the destination pixels of an alpha blended destination word;
storing the word components of said fourth word to an unaligned area of a memory in parallel.

2. The method of claim 1 wherein:

said step of alpha blending comprises the step of arithmetically combining corresponding sourcel, source2, and control pixels according to a predetermined formula to generate a corresponding destination pixel.

3. The method of claim 2 further comprising:

specifying a precision value for each of said source1, source2, control, and destination pixels;
reordering operations and terms of said predetermined formula to achieve the precision value and increase efficiency of said alpha blending step.

4. The method of claim 2 wherein:

said step of arithmetically combining utilizes predefined partitioned add, multiply, and subtract operations to operate on components of said first, second, and third words in parallel;
reordering operations and terms of said predetermined formula to increase the efficiency of operation of said predetermined partitioned operations.

5. In a computer system, a method of blending first and second source images to generate a destination image utilizing a control image where any one of said images is an unaligned image stored in a memory having boundaries unaligned with the addresses of said memory and where said images comprise words including multiple pixels, and with each pixel in said control image comprising a control pixel number of bits, said method comprising the steps of:

loading a first word from said first and second source images and said control image, and, if one of said images is an unaligned image;
generating an aligned address immediately preceding an unaligned address of a first word in said unaligned image;
calculating an offset being the difference between said aligned address and said unaligned address;
utilizing said unaligned address and said offset to load a word from said unaligned image;
expanding a subset of the pixels in the first word of said first and second source images into expanded pixels having equal numbers of leading and trailing zeros to form an expanded first word including said expanded pixels;
performing a partitioned subtraction operation to subtract corresponding expanded pixels in said first expanded words of said first and second source images to form an expanded difference word including expanded difference components;
performing a partitioned multiplication of corresponding pixels in said first word of said control image and said corresponding expanded difference components to form an expanded product word comprising expanded product components, with each expanded product component including the same number of leading zeros as said expanded pixel and having said control pixel number of least significant bits truncated to effect division by 2 raised to the power of said control pixel number;
performing a partitioned sum of said expanded product word and said first expanded word first source image to form a first expanded halfword of said destination image comprising expanded destination components;
packing said destination components of said expanded halfword to form a subset of the pixels of said destination word.

6. In a computer system, a method of blending first and second source images to generate a destination image

utilizing a control image where any one of said images is an unaligned image stored in a memory having boundaries unaligned with the addresses of said memory and where said images comprise words including multiple pixels, and with each pixel in said control image comprising a control pixel number of bits, said method comprising the steps of:

loading a first word from said first and second source images and said control image, and, if one of said images is an unaligned image;
generating an aligned address immediately preceding an unaligned address of a first word in said unaligned image;
calculating an offset being the difference between said aligned address and said unaligned address;
utilizing said unaligned address and said offset to load a word from said unaligned image;
defining first and second constants equal to 0x80008000 and 0x00ff00ff respectively;
performing a partitioned subtraction operation to subtract corresponding components of said first constant from said components in said first word of said control image in parallel to form a first difference word;
returning an aligned data word comprising components of said second constant and components of said first difference word;
performing a logical AND operation of corresponding components of said aligned data word and said second constant to generate a logical result word;
performing a partitioned packing operation of said logical result word to form a packed logical result word;
performing a partitioned multiplication of said first word of said control image and said first word of said first source image to form a first resulting product word;
performing a partitioned multiplication of said packed logical result word said first word of said first source image to form a second resulting product word;
performing a partitioned add operation on said first and second resulting product words to form a first sum word;
performing a partitioned multiplication of said first word of said control image and said first word of said second source image to form a third resulting product word;
performing a partitioned multiplication of said packed logical result word said first word of said second source image to form a fourth resulting product word;
performing a partitioned add operation on said third and fourth resulting product words to form a second sum word;
performing a partitioned add operation of said first and second sum words to form said first destination word;
computing an edge mask to store said destination word to said unaligned destination image;
utilizing said edge mask to perform a partial store of said destination word to said unaligned destination image.

7. An apparatus for forming a composite image by blending two digital images, the apparatus characterised by:

means for loading a first word into a data processor, the first word comprising a plurality of word components, each word component associated with a source 1 pixel of a first source image;
means for loading a second word into the data processor, the second word comprising plurality of word components, each word component associated with a source 2 pixel of a second source image;
means for loading a third word into the data processor, the third word comprising plurality of word components, each word component associated with a control pixel of a control image;
means for blending the components of said first, second, and third words in parallel to generate word components of a fourth word, with the word components of said fourth word associated with the destination pixels of a blended destination word: and
means for storing the word components of said fourth word to an unaligned area of a memory in parallel.

8. An apparatus as claimed in claim 9, characterised in that the composite image is a destination image and any one of said images is an unaligned image stored in a memory having boundaries unaligned with the addresses of said memory and where said images comprise words including multiple pixels, and with each pixel in said control image comprising a control pixel number of bits, and wherein if one of said images is an unaligned image the apparatus is further arranged to generate an aligned address immediately preceding an unaligned address of a first word in said unaligned image to calculate an offset being the difference between said aligned address and said unaligned address; and to utilize said unaligned address and said offset to load a word from said unaligned image.

9. An apparatus as claims in claims 9 and 10, characterised by being further arranged to expand a subset of the pixels in the first word of said first and second source images into expanded pixels having equal numbers of leading and trailing zeros to form an expanded first word including said expanded pixels;

to perform a partitioned subtraction operation to subtract corresponding expanding pixels in said first expanded words of said first and second source images to form an expanded difference word including expanded difference components;
to perform a partitioned multiplication of corresponding pixels in said first word of said control image and said corresponding expanded difference components to form an expanded product word comprising expanded product components, with each expanded product component including the same number of leading zeros as said expanded pixel and having said control pixel number of least significant bits truncated to effect division by 2 raised to the power of said control pixel number;
to perform a partitioned sum of said expanded product word and said first expanded word first source image to form a first expanded halfword of said destination image comprising expanded destination components; and
to pack said destination components of said expanded halfword to form a subset of the pixels of said destination word.

10. An apparatus as claimed in claim 7, characterised in that any one of said images is an unaligned image stored in a memory having boundaries unaligned with the addresses of said memory and where said images comprise words including multiple pixels, and with each pixel in said control image comprising a control pixel number of bits, the apparatus being arranged if one of said images is an unaligned image;

to generate an aligned address immediately preceding an unaligned address of a first word in said unaligned image;
to calculate an offset being the difference between said aligned address and said unaligned address;
to utilize said unaligned address and said offset to load a word from said unaligned image.

11. An apparatus as claimed in claims 9 or 12, characterised in being arranged to define first and second constants equal to 0x80008000 and 0x00ff00ff respectively;

to perform a partitioned subtraction operation to subtract corresponding components of said first constant from said components in said first word of said control image in parallel to form a first difference word;
to return an aligned data word comprising components of said second constant and components of said first difference word;
to perform a logical AND operation of corresponding components of said aligned data word and said second constant to generate a logical result word;
to perform a partitioned packing operation of said logical result word to form a packed logical result word;
to perform a partitioned multiplication of said first word of said control image and said first word of said first source image to form a first resulting product word;
to perform a partitioned multiplication of said packed logical result word said first word of said first source image to form a second resulting product word;
to perform a partitioned add operation on said first and second resulting product words to form a first sum word;
to perform a partitioned multiplication of said first word of said control image and said first word of said second source image to form a third resulting product word;
to perform a partitioned multiplication of said packed logical result word said first word of said second source image to form a fourth resulting product word;
to perform a partitioned add operation on said third and fourth resulting product words to form a second sum word;
to perform a partitioned add operation of said first and second sum words to form said first destination word;
to compute an edge mask to store said destination word to said unaligned destination image; and
to utilize said edge mask to perform a partial store of said destination word to said unaligned destination image.

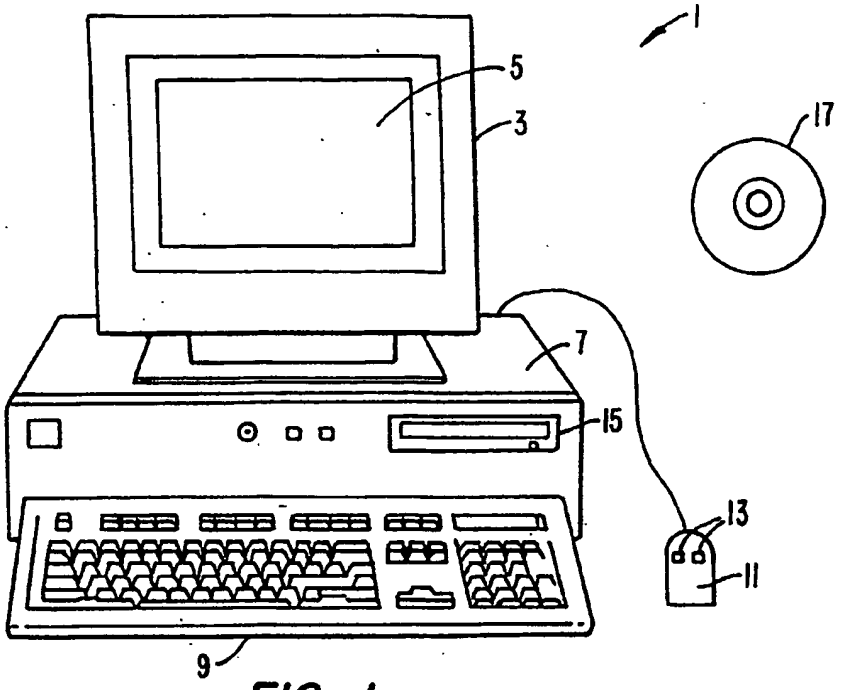

1
5
3
17
7
15
13
11
9

FIG. 1.

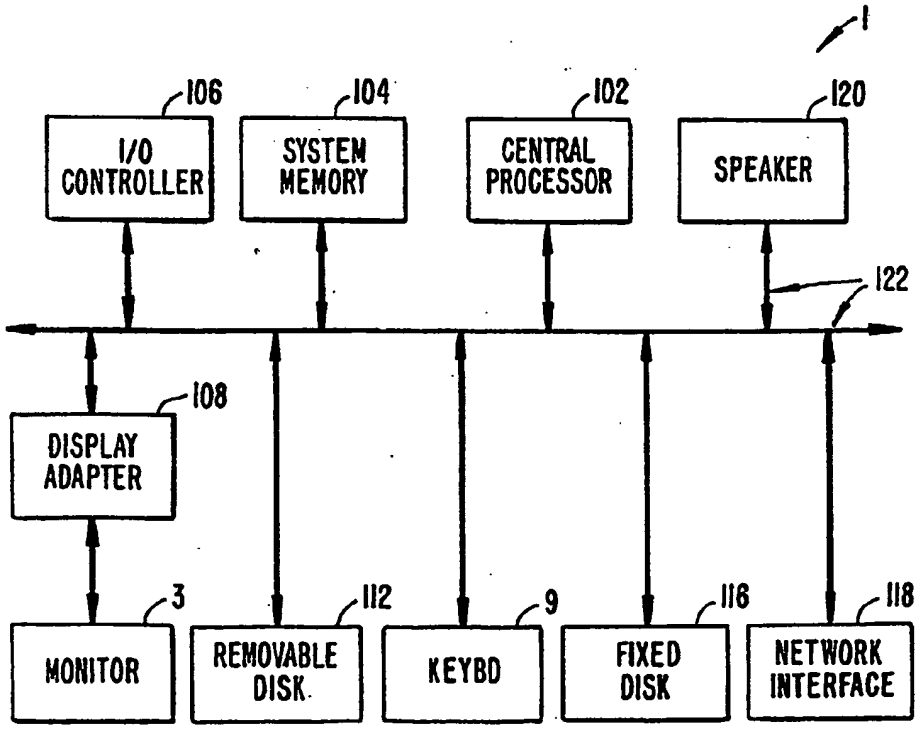

1
106
I/O CONTROLLER
104
SYSTEM MEMORY
102
CENTRAL PROCESSOR
120
SPEAKER
122
108
DISPLAY ADAPTER
3
MONITOR
112
REMOVABLE DISK
9
KEYBD
116
FIXED DISK
118
NETWORK INTERFACE

FIG. 2.

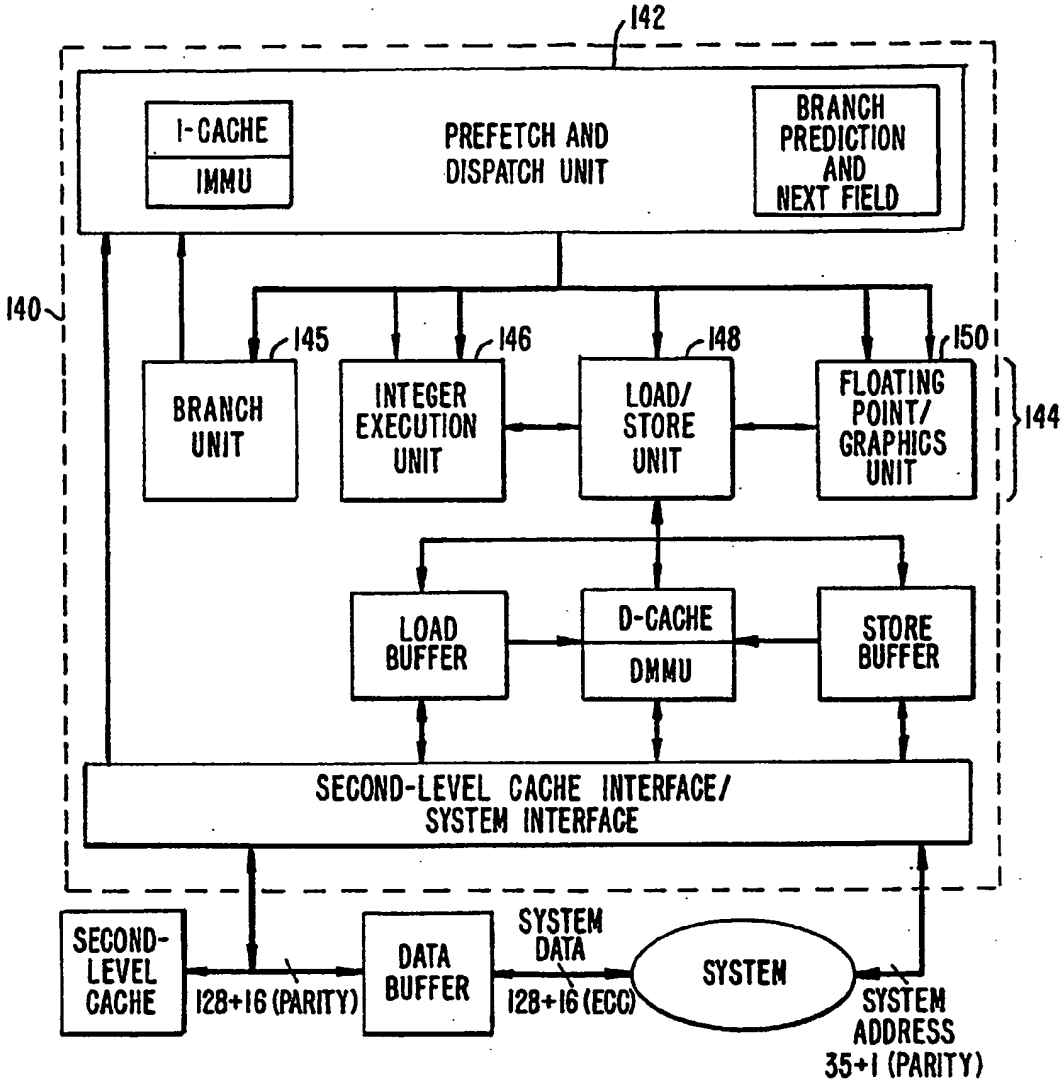

142
I-CACHE
IMMU
PREFETCH AND
DISPATCH UNIT
BRANCH
PREDICTION
AND
NEXT FIELD
140
145
146
148
150
BRANCH
UNIT
INTEGER
EXECUTION
UNIT
LOAD/
STORE
UNIT
FLOATING
POINT/
GRAPHICS
UNIT
144
LOAD
BUFFER
D-CACHE
DMMU
STORE
BUFFER
SECOND-LEVEL CACHE INTERFACE/
SYSTEM INTERFACE
SECOND-
LEVEL
CACHE
128+16 (PARITY)
DATA
BUFFER
SYSTEM
DATA
128+16 (ECC)
SYSTEM
SYSTEM
ADDRESS
35+1 (PARITY)

FIG. 3.

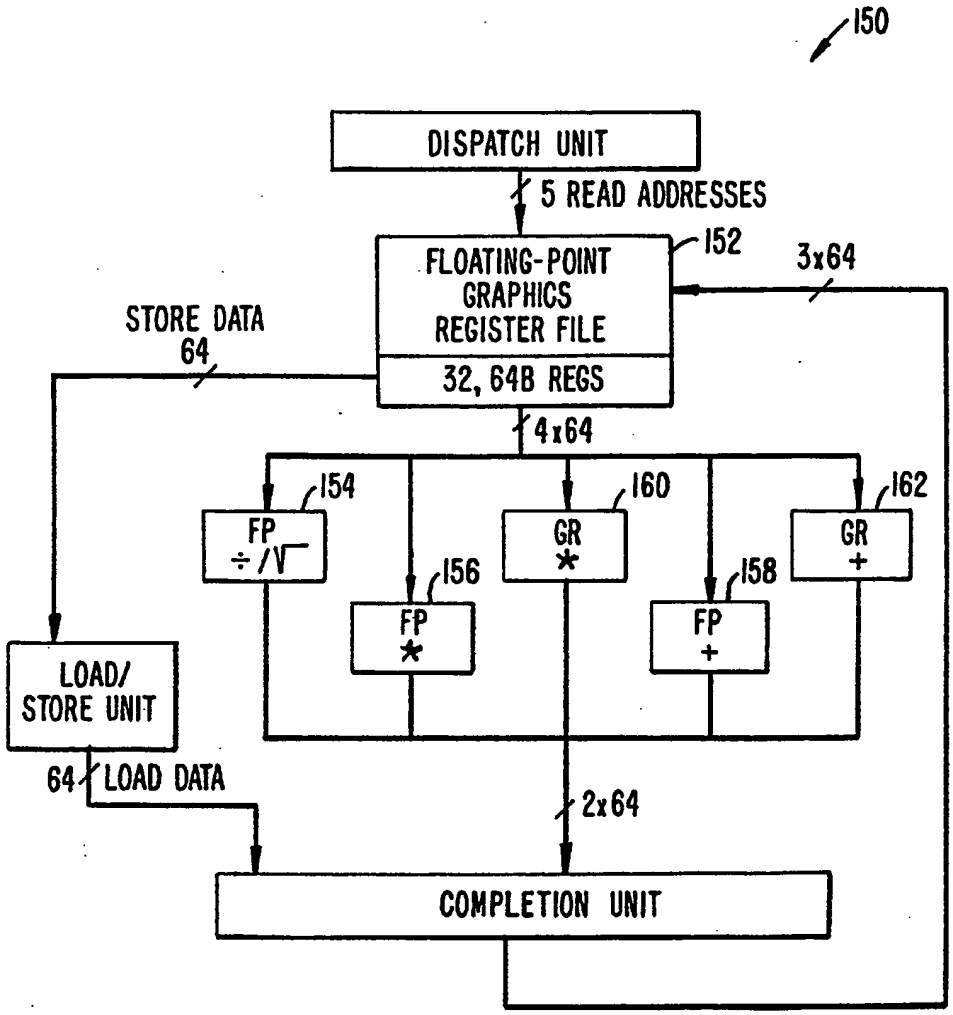

150
DISPATCH UNIT
5 READ ADDRESSES
FLOATING-POINT GRAPHICS REGISTER FILE
152
32, 64B REGS
3x64
STORE DATA 64
4x64
154
FP ÷/√
156
FP *
160
GR *
158
FP +
162
GR +
LOAD/ STORE UNIT
64
LOAD DATA
2x64
COMPLETION UNIT

FIG. 4.

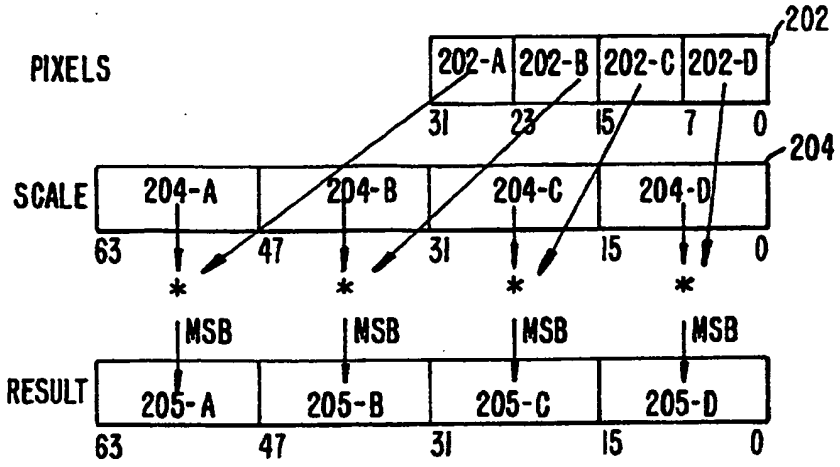

PIXELS
202-A
202-B
202-C
202-D
202
31
23
15
7
0
SCALE
204-A
204-B
204-C
204-D
204
63
47
31
15
0
MSB
MSB
MSB
MSB
RESULT
205-A
205-B
205-C
205-D
63
47
31
15
0

FIG. 5.

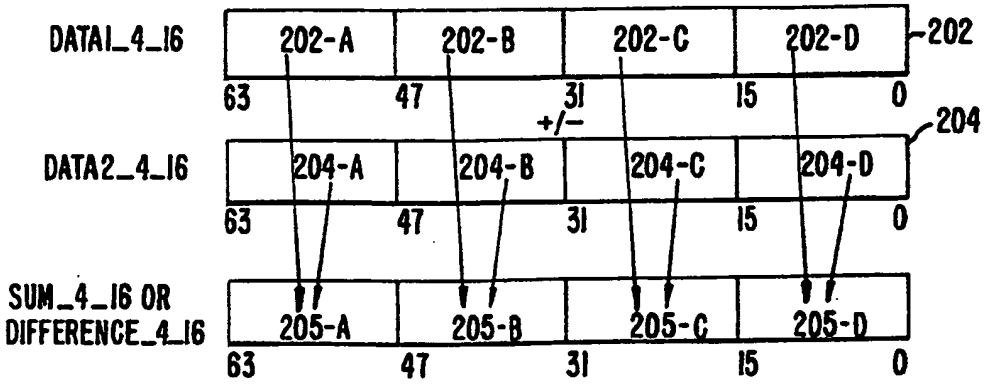

DATA1_4_16
202-A
202-B
202-C
202-D
202
63
47
31
15
0
+/−
DATA2_4_16
204-A
204-B
204-C
204-D
204
63
47
31
15
0
SUM_4_16 OR
DIFFERENCE_4_16
205-A
205-B
205-C
205-D
63
47
31
15
0

FIG. 6.

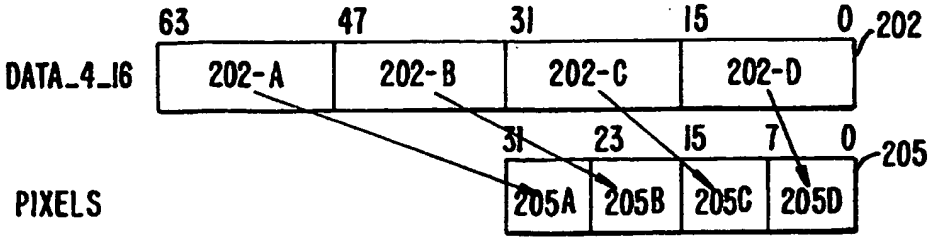


FIG. 7A.

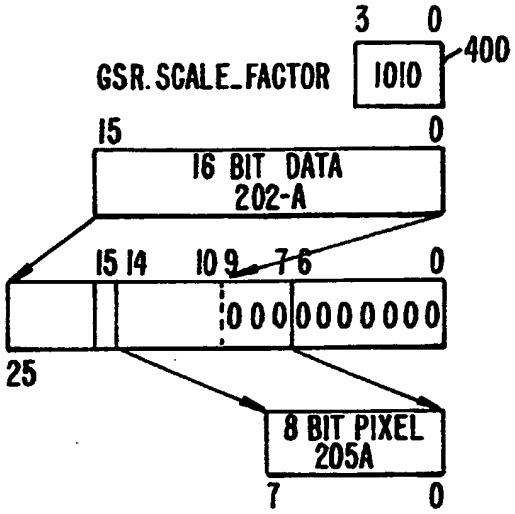


FIG. 7B.

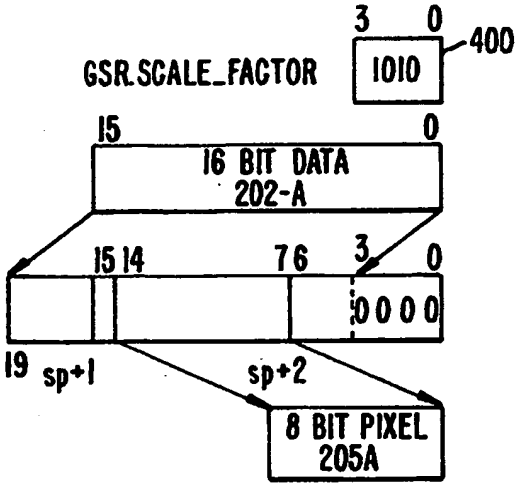


FIG. 7C.

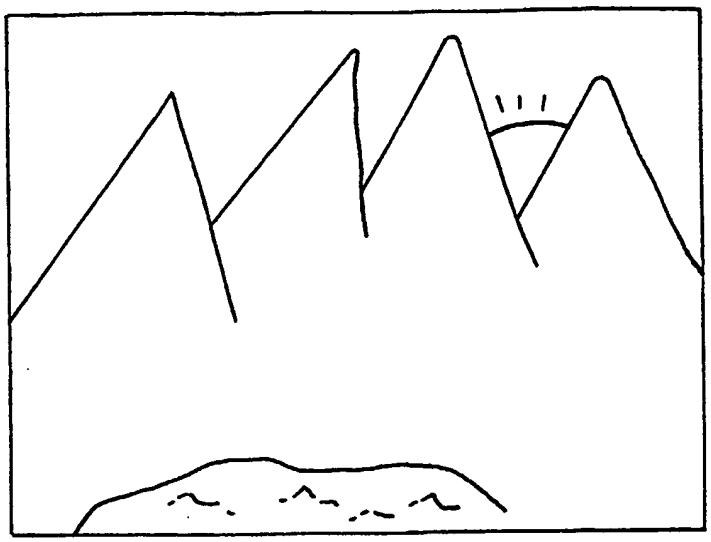


FIG. 8A.

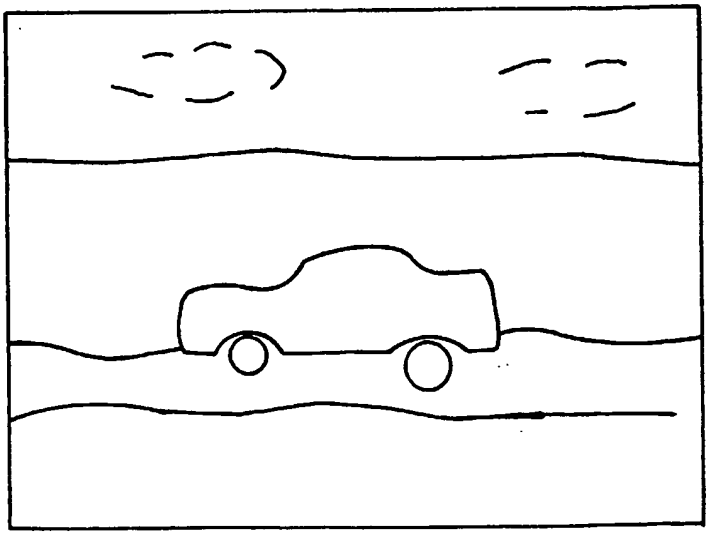


FIG. 8B.

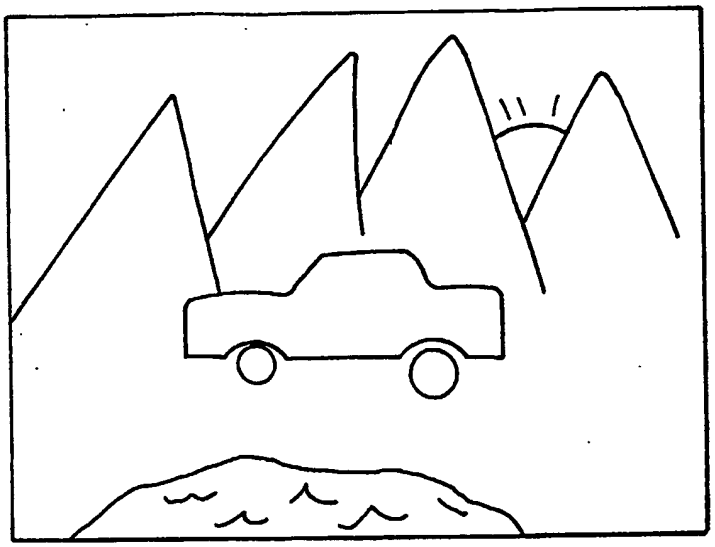


FIG. 8C.

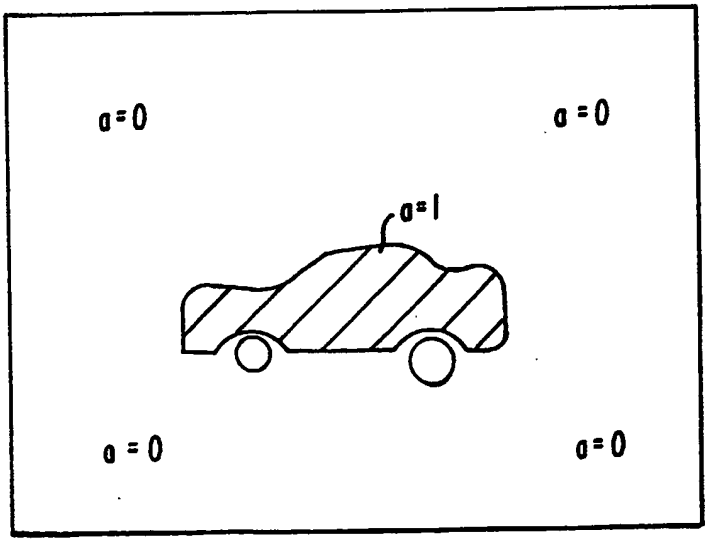


FIG. 8D.

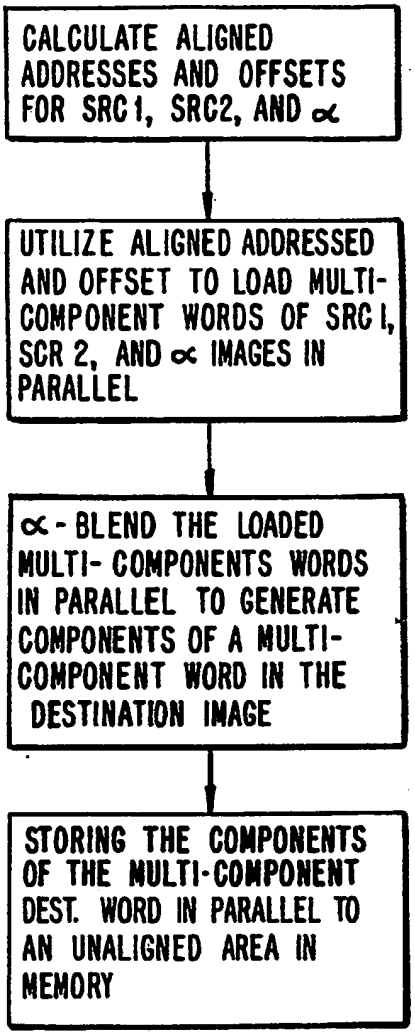

CALCULATE ALIGNED ADDRESSES AND OFFSETS FOR SRC1, SRC2, AND α
UTILIZE ALIGNED ADDRESSED AND OFFSET TO LOAD MULTI-COMPONENT WORDS OF SRC 1, SCR 2, AND α IMAGES IN PARALLEL
α - BLEND THE LOADED MULTI- COMPONENTS WORDS IN PARALLEL TO GENERATE COMPONENTS OF A MULTI-COMPONENT WORD IN THE DESTINATION IMAGE
STORING THE COMPONENTS OF THE MULTI-COMPONENT DEST. WORD IN PARALLEL TO AN UNALIGNED AREA IN MEMORY

FIG. 9.

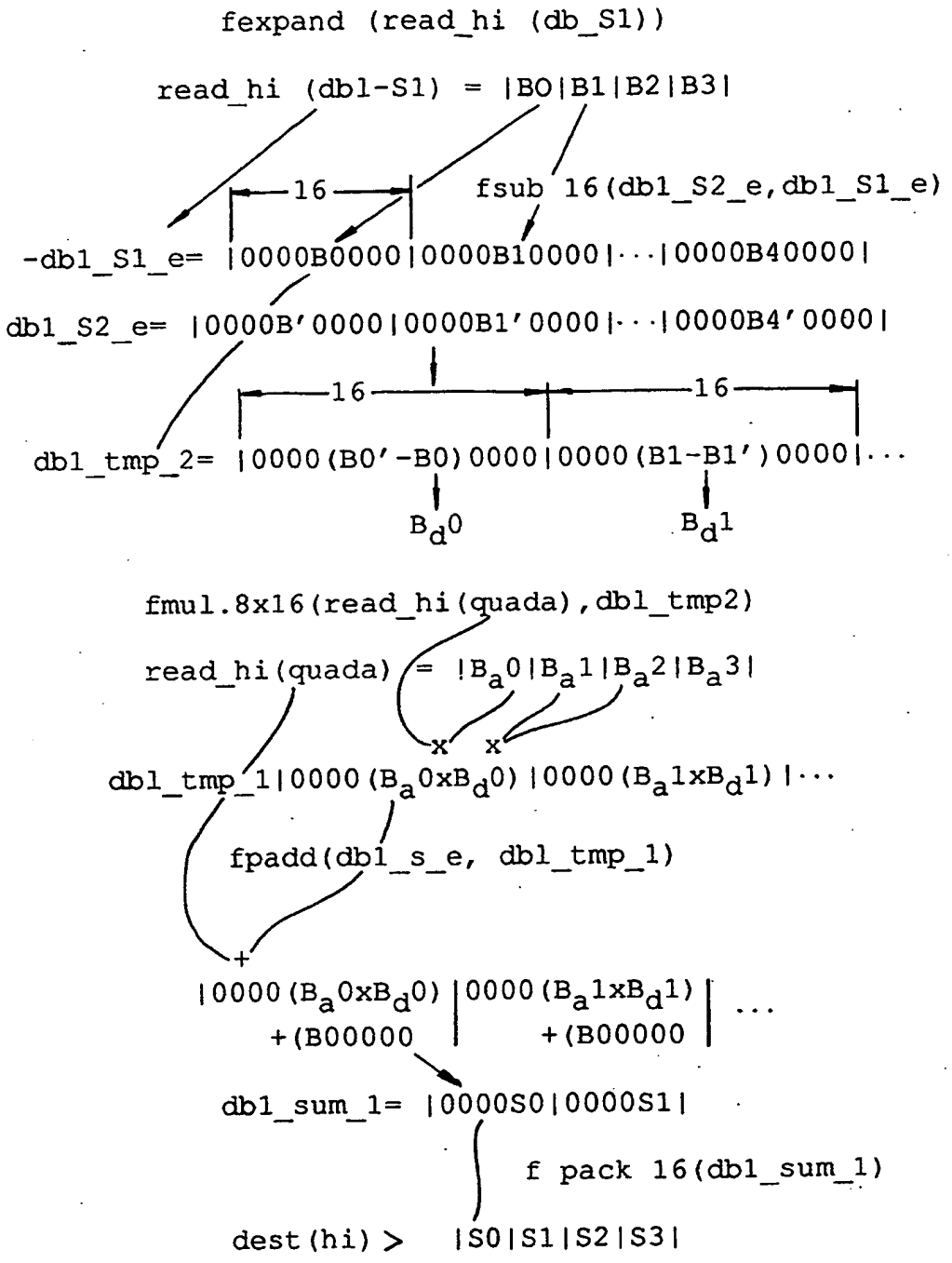

fexpand (read_hi (db_S1))
read_hi (dbl-S1) = |B0|B1|B2|B3|
16
fsub 16(db1_S2_e,db1_S1_e)
-db1_S1_e= |0000B0000|0000B10000|···|0000B40000|
db1_S2_e= |0000B'0000|0000B1'0000|···|0000B4'0000|
16
16
db1_tmp_2= |0000(B0'-B0)0000|0000(B1-B1')0000|···
B_d0
B_d1
fmul.8x16(read_hi(quada),dbl_tmp2)
read_hi(quada) = |B_a0|B_a1|B_a2|B_a3|
x
x
dbl_tmp_1|0000(B_a0xB_d0)|0000(B_a1xB_d1)|···
fpadd(db1_s_e, dbl_tmp_1)
+
|0000(B_a0xB_d0)|0000(B_a1xB_d1)| ...
+(B00000
+(B00000
db1_sum_1= |0000S0|0000S1|
f pack 16(db1_sum_1)
dest(hi) > |S0|S1|S2|S3|

FIG. 10.

(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) EP 0 790 581 A3

(12) EUROPEAN PATENT APPLICATION

(88) Date of publication A3:
29.12.1997 Bulletin 1997/52

(43) Date of publication A2:
20.08.1997 Bulletin 1997/34

(21) Application number: **96308513.9**

(22) Date of filing: **26.11.1996**

(51) Int Cl.[6]: **G06T 15/10**

(84) Designated Contracting States:
GB SE

(30) Priority: **27.11.1995 US 563033**

(71) Applicant: **SUN MICROSYSTEMS, INC.**
Mountain View, CA 94043 (US)

(72) Inventor: **Hu, Xiao Ping**
Mountain View, California 94043 (US)

(74) Representative: **Hogg, Jeffery Keith et al**
Withers & Rogers
4 Dyer's Buildings
Holborn
London EC1N 2JT (GB)

(54) Method for alpha blending images utilizing a visual instruction set

(57) An image alpha blending method utilizing a parallel processor is provided. The computer-implemented method includes the steps of loading unaligned multiple word components into a processor in one machine instruction, each word component associated with a pixel of an image; alpha blending the multiple word components of different source images and a control image in parallel; and storing the alpha blended multiple word components of a destination image into memory in parallel.

EP 0 790 581 A3

Printed by Jouve, 75001 PARIS (FR)

European Patent Office

EUROPEAN SEARCH REPORT

Application Number
EP 96 30 8513

DOCUMENTS CONSIDERED TO BE RELEVANT

Category	Citation of document with indication, where appropriate, of relevant passages	Relevant to claim	CLASSIFICATION OF THE APPLICATION (Int.Cl.6)
X	GB 2 278 524 A (NIHON UNISYS LTD) 30 November 1994 * page 16, line 5 - line 25 * * page 18, line 11 - line 16 * * page 23, line 8 - line 12 * * page 24, line 23 - line 29 * * page 28, line 9 - page 32, line 32 *	1,2,4-8	G06T15/10
A	EP 0 465 250 A (CANON KK ;CANON INFORMATION SYST RES (AU)) 8 January 1992 * page 5, line 8 - line 30 * * page 12, line 13 - page 13, line 50 *	1,2,5-7	
A	WATANABE T ET AL: "3-D CG MEDIA CHIP: AN EXPERIMENTAL SINGLE-CHIP ARCHITECTURE FOR THREE-DIMENSIONAL COMPUTER GRAPHICS" IEICE TRANSACTIONS ON ELECTRONICS, vol. E77-C, no. 12, 1 December 1994, pages 1881-1887, XP000497019 * page 1882, right-hand column, line 4 - page 1883, left-hand column, line 29 * * page 1884, left-hand column, line 2 - page 1885, right-hand column, line 34 *	1,2,5-7	

TECHNICAL FIELDS SEARCHED (Int.Cl.6)
G06T

The present search report has been drawn up for all claims

Place of search	Date of completion of the search	Examiner
BERLIN	16 October 1997	Burgaud, C

CATEGORY OF CITED DOCUMENTS

X : particularly relevant if taken alone
Y : particularly relevant if combined with another document of the same category
A : technological background
O : non-written disclosure
P : intermediate document

T : theory or principle underlying the invention
E : earlier patent document, but published on, or after the filing date
D : document cited in the application
L : document cited for other reasons

& : member of the same patent family, corresponding document

EPO FORM 1503 03.82 (P04C01)